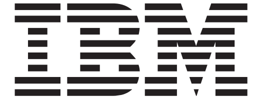
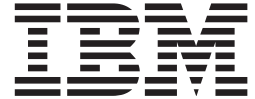
IBM® DB2 Universal Database™



SQL Reference for Cross-Platform Development

Version 1

IBM® DB2 Universal Database™



SQL Reference for Cross-Platform Development

Version 1

Before using this information and the products it supports, be sure to read the general information under "Notices" on page 695.

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

© Copyright International Business Machines Corporation 1982, 2002. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book ix

Who Should Read This Book. ix
How to Use This Book ix
  Assumptions Relating to Examples of SQL Statements x
  How to Read the Syntax Diagrams x
  Conventions for Describing Mixed Data Values xii
SQL Accessibility xii
Related Documentation xiii
  DB2 Universal Database for z/OS and OS/390 xiii
  DB2 Universal Database for iSeries xiii
  DB2 Universal Database for the Unix and Windows and OS/2 Platforms xiii
  Distributed Relational Database Architecture . . xiii
  Character Data Representation Architecture . . xiv
  Industry Standards xiv

Chapter 1. Concepts 1

Relational Database 1
Structured Query Language 1
  Static SQL 1
  Dynamic SQL 2
  Interactive SQL 2
  SQL Call Level Interface (CLI) and Open Database Connectivity (ODBC) 2
  Java Database Connectivity (JDBC) and Embedded SQL for Java (SQLJ) Programs 2
Schemas 3
Tables 3
Keys 3
Primary Keys and Unique Keys 3
Referential Integrity 4
Check Constraints 6
Triggers 6
Indexes 8
Views. 8
Aliases 8
Packages and Access Plans 9
Routines 9
  Functions 9
  Procedures 9
Catalog 11
Application Processes, Concurrency, and Recovery 11
Isolation Level 13
  Repeatable Read (RR) 14
  Read Stability (RS) 14
  Cursor Stability (CS) 15
  Uncommitted Read (UR) 15
  Comparison of Isolation Levels 15
Distributed Relational Database 16
  Application Servers. 17
  CONNECT (Type 1) and CONNECT (Type 2) . . 17
  Remote Unit of Work 18
  Application-Directed Distributed Unit of Work 20
  Data Representation Considerations 23
Character Conversion 23
  Character Sets and Code Pages 24
  Coded Character Sets and CCSIDs. 26
  Default CCSID 26
Authorization, Privileges and Object Ownership . . 27
Storage Structures 28

Chapter 2. Language Elements 29

Characters 29
Tokens 30
Identifiers 31
  SQL Identifiers 31
  Host Identifiers 31
Naming Conventions 32
  SQL Path 38
  Qualification of Unqualified Object Names . . . 38
Aliases 40
Authorization IDs and Authorization Names . . . 41
  Example 41
Data Types 42
  Nulls 43
  Numbers 43
  Character Strings 44
  Character Encoding Schemes 44
  Graphic Strings 45
  Graphic Encoding Schemes 46
  Binary Strings 46
  Large Objects (LOBs) 47
  Limitations on Use of Strings 48
  Datetime Values 49
  User-Defined Types. 51
Promotion of Data Types 53
Casting Between Data Types. 54
Assignments and Comparisons 58
  Numeric Assignments 59
  String Assignments 60
  Datetime Assignments. 62
  Distinct type Assignments 63
  Assignments Involving LOB Locators. 64
  Numeric Comparisons. 64
  String Comparisons. 65
  Datetime Comparisons 66
  Distinct type Comparisons 67
Rules for Result Data Types 68
  Numeric Operands 68
  Character String Operands 69
  Graphic String Operands 69
  Binary String Operands 69
  Datetime Operands. 70
  Distinct Type Operands 70
Conversion Rules for Operations that Combine Strings 71
Constants 73
  Integer Constants 73
  Floating-Point Constants 73

© Copyright IBM Corp. 1982, 2002

Decimal Constants 73
Character-String Constants 73
Graphic-String Constants 74
Decimal Point 75
Special Registers. 76
CURRENT DATE 76
CURRENT PATH 76
CURRENT SERVER 77
CURRENT TIME 77
CURRENT TIMESTAMP 77
CURRENT TIMEZONE 78
USER 78
Column Names 79
Qualified Column Names. 79
Correlation Names 79
Column Name Qualifiers to Avoid Ambiguity . . 81
Column Name Qualifiers in Correlated References 83
Unqualified Column Names in Correlated References 84
References to Variables 85
References to Host Variables. 85
Host Variables in Dynamic SQL 87
References to LOB Host Variables 88
References to LOB Locator Variables 88
Host Structures 89
Functions 91
Types of Functions 91
Function Invocation 92
Function Resolution 92
Determining the Best Fit 93
Best Fit Considerations 95
Expressions 96
Without Operators 96
With Arithmetic Operators 96
With the Concatenation Operator 98
Datetime Operands and Durations 100
Datetime Arithmetic in SQL 101
Precedence of Operations 105
CASE Expressions 106
CAST Specification 109
Predicates 112
Basic Predicate 113
Quantified Predicate 114
BETWEEN Predicate 116
EXISTS Predicate 117
IN Predicate 118
LIKE Predicate 120
NULL Predicate 125
Search Conditions 126
Examples. 126

Chapter 3. Built-in Functions. 129
Column Functions. 133
AVG 134
COUNT 135
COUNT_BIG 136
MAX 137
MIN 138
STDDEV 139
SUM 140
VARIANCE or VAR 141
Scalar Functions 142
Example 142
ABS 143
ACOS 144
ASIN 145
ATAN 146
ATAN2 147
BLOB 148
CEILING 149
CHAR. 150
CLOB 155
COALESCE 156
CONCAT. 157
COS 158
DATE 159
DAY 161
DAYOFWEEK 162
DAYOFWEEK_ISO 163
DAYOFYEAR 164
DAYS 165
DBCLOB 166
DECIMAL or DEC 167
DEGREES 169
DIGITS 170
DOUBLE_PRECISION or DOUBLE 171
EXP 173
FLOAT 174
FLOOR 175
GRAPHIC 176
HEX 177
HOUR. 178
INTEGER or INT 179
JULIAN_DAY 180
LCASE 181
LEFT 182
LENGTH 183
LN 184
LOCATE 185
LOG10 187
LOWER 188
LTRIM 189
MICROSECOND 190
MIDNIGHT_SECONDS 191
MINUTE 192
MOD 193
MONTH 194
NULLIF 195
POSSTR 196
POWER 198
QUARTER 199
RADIANS 200
RAND. 201
REAL 202
ROUND 203
RTRIM 205
SECOND 206
SIGN 207
SIN. 208
SMALLINT 209
SPACE 210

SQRT 211
SUBSTR 212
TAN 214
TIME 215
TIMESTAMP 216
TRANSLATE 218
TRUNCATE or TRUNC 220
UCASE 222
UPPER 223
VALUE 224
VARCHAR 225
VARGRAPHIC 227
WEEK 230
WEEK_ISO 231
YEAR 232

Chapter 4. Queries 233
Authorization 233
subselect 234
select-clause 235
from-clause 239
where-clause 243
group-by-clause 244
having-clause 245
Examples of a subselect 246
fullselect 248
Rules for Columns 248
Examples of a fullselect 249
select-statement 250
order-by-clause 251
fetch-first-clause 253
update-clause 254
read-only-clause 255
optimize-clause 256
isolation-clause 257
Examples of a select-statement 258

Chapter 5. Statements 259
How SQL Statements Are Invoked 262
Embedding a Statement in an Application Program 262
Dynamic Preparation and Execution. 263
Static Invocation of a select-statement 264
Dynamic Invocation of a select-statement . . . 264
Interactive Invocation 264
SQL Return Codes. 264
SQLSTATE 265
SQLCODE 265
SQL Comments. 266
ALTER TABLE 267
BEGIN DECLARE SECTION 281
CALL 283
CLOSE 287
COMMENT 289
COMMIT. 294
CONNECT (Type 1) 296
CONNECT (Type 2) 300
CREATE ALIAS 303
CREATE DISTINCT TYPE 304
CREATE FUNCTION. 310
CREATE FUNCTION (External Scalar) 314
CREATE FUNCTION (Sourced) 325
CREATE FUNCTION (SQL Scalar) 332
CREATE INDEX 338
CREATE PROCEDURE 340
CREATE PROCEDURE (External) 341
CREATE PROCEDURE (SQL) 348
CREATE TABLE 353
CREATE TRIGGER 368
CREATE VIEW 376
DECLARE CURSOR 381
DELETE 386
DESCRIBE 391
DROP 395
END DECLARE SECTION 400
EXECUTE 401
EXECUTE IMMEDIATE 404
FETCH 406
FREE LOCATOR 409
GRANT (Distinct Type Privileges) 410
GRANT (Function or Procedure Privileges) . . . 412
GRANT (Package Privileges) 416
GRANT (Table or View Privileges) 418
INCLUDE 421
INSERT 423
LOCK TABLE 428
OPEN 429
PREPARE 433
RELEASE (Connection) 440
RENAME 442
REVOKE (Distinct Type Privileges) 444
REVOKE (Function or Procedure Privileges) . . . 446
REVOKE (Package Privileges) 450
REVOKE (Table and View Privileges) 452
ROLLBACK 455
SELECT 456
SELECT INTO 457
SET CONNECTION 459
SET PATH 461
SET transition-variable 464
UPDATE 466
VALUES 472
VALUES INTO 473
WHENEVER 475

Chapter 6. SQL Control Statements 477
References to SQL Parameters and SQL Variables 478
SQL-procedure-statement 479
assignment-statement. 480
CALL Statement 481
CASE Statement 482
compound-statement 484
GET DIAGNOSTICS Statement 491
GOTO Statement 493
IF Statement. 494
LEAVE Statement 496
LOOP Statement 497
REPEAT Statement 498
RESIGNAL Statement 500
RETURN Statement 502
SIGNAL Statement 504

WHILE Statement 506

Appendix A. SQL Limits 509

Appendix B. Characteristics of SQL Statements 515

Actions Allowed on SQL Statements. 516
SQL Statement Data Access Classification for Routines 518
Considerations for Using Distributed Relational Database 520

Appendix C. SQL Communication Area (SQLCA) 525

Field Descriptions 525
INCLUDE SQLCA Declarations 527
  For C 527
  For COBOL 527

Appendix D. SQL Descriptor Area (SQLDA) 529

Field Descriptions in an SQLDA Header 530
  Determining How Many Occurrences of SQLVAR Entries are Needed 531
Field Descriptions in an Occurrence of SQLVAR 532
SQLTYPE and SQLLEN 533
CCSID Values in SQLDATA and SQLNAME . . . 535
INCLUDE SQLDA Declarations 536
  For C 536
  For COBOL 538

Appendix E. SQLSTATE Values—Common Return Codes . . . 539

Using SQLSTATE Values 539

Appendix F. CCSID Values 575

Appendix G. CONNECT (Type 1) and CONNECT (Type 2) Differences. . . . 591

Determining the CONNECT Rules That Apply . . 591
Connecting to Application Servers That Only Support Remote Unit of Work 592

Appendix H. Coding SQL Statements in C Applications 593

Defining the SQL Communications Area in C . . 593
Defining SQL Descriptor Areas in C 593
Embedding SQL Statements in C 595
  Comments 595
  Continuation for SQL Statements 595
  Including Code. 595
  Margins 596
  Names 596
  NULLs and NULs 596
  Statement Labels 596
  Preprocessor Considerations 596
  Trigraphs 596
  Handling SQL Errors and Warnings in C . . . 597
Using Host Variables in C 597
  Declaring Host Variables in C 597
  Declaring Host Structures in C 603
  Using Pointer Data Types in C 606
Determining Equivalent SQL and C Data Types 606

Appendix I. Coding SQL Statements in COBOL Applications 609

Defining the SQL Communications Area in COBOL 609
Defining SQL Descriptor Areas in COBOL. . . . 609
Embedding SQL Statements in COBOL. 610
  Comments 610
  Continuation for SQL statements 611
  Cursors 611
  Including Code. 611
  Margins 611
  Names. 611
  Statement Labels 611
  Handling SQL Errors and Warnings in COBOL 612
Using Host Variables in COBOL 612
  Declaring Host Variables in COBOL 612
  Declaring Host Structures in COBOL 620
Determining Equivalent SQL and COBOL Data Types 623
  Notes on COBOL Variable Declaration and Usage 625

Appendix J. Coding SQL Statements in Java Applications 627

Defining the SQL Communications Area in Java 627
Defining SQL Descriptor Areas in Java 627
Embedding SQL Statements in Java 627
  Comments 628
Connecting To, and Using a Data Source 629
  Declaring a Connection Context 629
  Initiating and Using a Connection 630
Using Host Variables and Expressions in Java . . 631
Using SQLJ Iterators to Retrieve Rows From a Result Table 632
  Declaring Iterators. 633
  Using Positioned Iterators to Retrieve Rows From a Result Table 635
  Using Named Iterators to Retrieve Rows From a Result Table 636
Using Iterators For Positioned UPDATE and DELETE Operations 637
Handling SQL Errors and Warnings in Java . . . 639
Determining Equivalent SQL and Java Data Types 640
Example 641

Appendix K. Coding SQL Statements in REXX Applications 643

Defining the SQL Communications Area in REXX 644
Defining SQL Descriptor Areas in REXX 644
Embedding SQL Statements in REXX 646
  Comments 646
  Continuation of SQL Statements 646
  Including Code. 646
  Margins 647
  Names 647

Nulls 647
Statement Labels 647
Handling SQL Errors and Warnings in REXX 647
Isolation Level 647
Using Host Variables in REXX. 648
Determining Data Types of Input Host Variables 648
The Format of Output Host Variables 649
Avoiding REXX Conversion 649
Indicator Variables in REXX 649
Example 650

Appendix L. Coding Programs for use by External Routines 653

Parameter Passing for External Routines 653
Parameter Passing for External Functions Written in C or COBOL 653
Parameter Passing for External Functions Written in Java 656
Parameter Passing for External Procedures Written in C or COBOL 657
Parameter Passing for External Procedures Written in Java 660
Attributes of the Arguments Passed to a Routine Program 660
Database Information in External Routines (DBINFO) 662
CCSID Information in DBINFO 664
DBINFO Structure for C. 664
DBINFO Structure for COBOL. 665
Scratch Pad in External Functions 666

Appendix M. Sample Tables 667

ACT 667
CL_SCHED 668
DEPARTMENT. 668
EMP_PHOTO 668
EMP_RESUME 669
EMPLOYEE 669
EMPPROJACT 672
IN_TRAY. 674
ORG 674
PROJACT 675
PROJECT. 676
SALES. 678
STAFF. 679
Sample Files With BLOB and CLOB Data Type . . 680
Quintana Photo 680
Quintana Resume 681
Nicholls Photo 682
Nicholls Resume 683
Adamson Photo 684
Adamson Resume 685
Walker Photo 686
Walker Resume. 687

Appendix N. Terminology Differences 689

Reserved Schema Names and Reserved Words 691

Reserved Schema Names 691
Reserved Words 691

Notices 695

Trademarks and Service Marks 695

Index 697

About This Book

This book defines IBM DB2 Universal Database Structured Query Language (DB2 UDB SQL). It describes the rules and limits for preparing portable programs. This book is a reference rather than a tutorial and assumes a familiarity with SQL programming concepts.

Who Should Read This Book

This book is intended for programmers who want to write portable applications using SQL that is common to the DB2 UDB relational database products and the SQL 1999 Core standard.[1] DB2 UDB SQL is consistent with the SQL 1999 Core standard. DB2 UDB SQL also provides functional extensions to the SQL 1999 Core standard. For example, many of the scalar functions defined in this book are extensions to the SQL 1999 Core standard.

How to Use This Book

This book defines the DB2 UDB SQL language elements that are common to the IBM DB2 Universal Database Family of relational database products across the following environments:

Environment	IBM Relational Database Product	Short Name
z/OS and OS/390	DB2 Universal Database for z/OS and OS/390 Version 7	DB2 UDB for z/OS and OS/390
OS/400	DB2 Universal Database for iSeries Version 5 Release 1	DB2 UDB for iSeries
UNIX • AIX • HP-UX Version 10 • HP-UX Version 11 • Linux • Solaris Windows for 32-bit operating systems OS/2	DB2 Universal Database for the Unix and Windows and OS/2 Platforms Version 7 Release 2	DB2 UDB for UWO

The DB2 Universal Database relational database products have product books that also describe product-specific elements and explain how to prepare and run a program in a particular environment. The information in this book is a subset of the information in the product books, and the rules and limits described in this book might not apply to all products. The limits in this book are those required to assist program portability across the applicable IBM environments. See "Related Documentation" on page xiii for a list of the product books needed in addition to this book.

The SQL described in this book assumes that default environment options, including:

- precompile options

1. In this book, the term "SQL 1999 Core standard" is used to describe the ANSI/ISO Core Level SQL standard of 1999 and related industry standards. See "Related Documentation" on page xiii for a list of documentation that describes these industry standards.

© Copyright IBM Corp. 1982, 2002

- bind options
- registry variables

are set to default values, unless specifically mentioned.

Since each DB2 Universal Database product does not ship on the same schedule, at any point in time there will naturally be some elements that are only available on a subset of the DB2 Universal Database products. Some elements might be implemented by all products, but differ slightly in their semantics (how they behave when the program is run). In many cases, these semantic difference are the result of the underlying operating system support. These conditions are identified in this book as shown in the next paragraph.

G G G G The DB2 UDB SQL definition is described in this book. If the implementation of an element in a product differs from the DB2 UDB SQL definition in its syntax or semantics, the difference is highlighted with a symbol in the left margin as is this sentence.

Assumptions Relating to Examples of SQL Statements

The examples of SQL statements shown in this guide are based on the sample tables in Appendix M, "Sample Tables" on page 667 and assume the following:

- 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. For more information see Appendix M, "Sample Tables" on page 667.

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 the Syntax Diagrams

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

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

  The ►►─── symbol indicates the beginning of a statement.

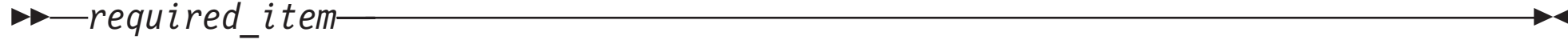
  The ───► symbol indicates that the statement syntax is continued on the next line.

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

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

  Diagrams of syntactical units other than complete statements start with the ►─── symbol and end with the ───► symbol.

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



- Optional items appear below the main path.

```
►►—required_item—┬───────────────┬——————————————————————————————►◄
                 └─optional_item─┘
```

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

```
                 ┌─optional_item─┐
►►—required_item—┴───────────────┴——————————————————————————————►◄
```

- If more than one item can be chosen, they appear vertically, in a stack.

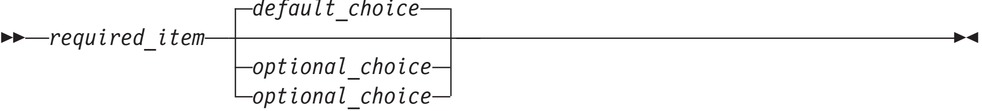
If one of the items must be chosen, one item of the stack appears on the main path.

```
►►—required_item——┬─required_choice1─┬——————————————————————————►◄
                  └─required_choice2─┘
```

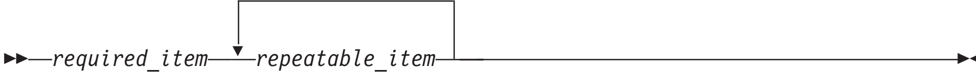
If choosing one of the items is optional, the entire stack appears below the main path.

```
►►—required_item—┬──────────────────┬———————————————————————————►◄
                 ├─optional_choice1─┤
                 └─optional_choice2─┘
```

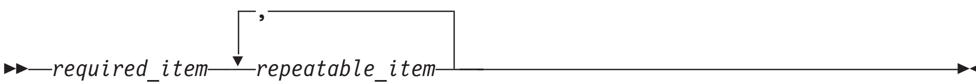
If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



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

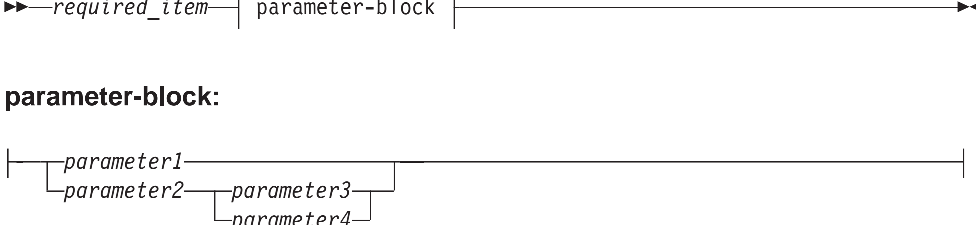


If the repeat arrow contains a comma, repeated items must be separated with a comma.



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

- 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, they must be entered 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.
- Sometimes a single variable represents a larger fragment of the syntax. For example, in the following diagram, the variable parameter-block represents the whole syntax fragment that is labeled **parameter-block**:



Conventions for Describing Mixed Data Values

When mixed data values are shown in the examples, the following conventions apply[2] :

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

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.

2. Hexadecimal values are for EBCDIC characters.

- The DB2 Universal Database products provide Windows graphical user interfaces to DB2 Universal Databases. 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 visit http://www-3.ibm.com/able/hpr.html.

The IBM Home Page Reader can be used to access all descriptive text in this book.

Related Documentation

The following documentation for DB2 Universal Database is available on the internet at:

http://www.software.ibm.com/data/db2

DB2 Universal Database for z/OS and OS/390

- *DB2 Universal Database for OS/390 and z/OS SQL Reference*
- *DB2 Universal Database for OS/390 and z/OS Application Programming and SQL Guide*
- *DB2 Universal Database for OS/390 and z/OS V7 Application Programming Guide and Reference for Java™*

DB2 Universal Database for iSeries

- *DB2 Universal Database for iSeries SQL Reference*
- *DB2 Universal Database for iSeries SQL Programming Concepts*
- *DB2 Universal Database for iSeries SQL Programming with Host Languages*
- *IBM Developer Kit for Java topic in the iSeries Information Center*
- *iSeries Information Center*, at http://publib.boulder.ibm.com/html/as400/infocenter.html

DB2 Universal Database for the Unix and Windows and OS/2 Platforms

- *IBM DB2 Universal Database SQL Reference*, Volumes 1 and 2
- *IBM DB2 Universal Database Application Development Guide*
- *IBM DB2 Universal Database Application Building Guide*

Distributed Relational Database Architecture

- *Application Programming Guide*, SC26-4773
- *Connectivity Guide*, SC26-4783
- *Open Group Publications: DRDA Vol. 1: Distributed Relational Database Architecture (DRDA)*, at http://www.opengroup.org/publications/catalog/c812.htm.
- *Open Group Publications: DRDA Vol. 2: Formatted Data Object Content Architecture (FD:OCA)*, at http://www.opengroup.org/publications/catalog/c813.htm.
- *Open Group Publications: DRDA Vol. 3: Distributed Data Management (DDM) Architecture*, at http://www.opengroup.org/publications/catalog/c814.htm.

Character Data Representation Architecture

- *Character Data Representation Architecture Reference and Registy*, SC09-2190

Industry Standards

- *Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework) ISO/IEC 9075-1:1999*
- *Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation) ISO/IEC 9075-2:1999*
- *Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM) ISO/IEC 9075-4:1999*
- *Information technology - Database languages - SQL - Part 5: Host Language Bindings (SQL/Bindings) ISO/IEC 9075-5:1999*
- *Information technology - Database languages - SQL - Part 10: Object Language Bindings (SQL/OLB) ISO/IEC 9075-10:2000*
- *Information technology - Database languages - SQL - Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT) ISO/IEC 9075-13:2002*
- *ANSI (American National Standards Institute) X3.135-1999, Database Language SQL*

Chapter 1. Concepts

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 is one relational database in:

- A DB2 UDB for z/OS and OS/390 subsystem
- An iSeries system.

In DB2 UDB for UWO, any number of relational databases can be created on a given physical machine.

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 one of the supported host languages; COBOL, C (C also covers C++ in this documentation, unless otherwise mentioned explicitly) or Java. 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 invoke the database manager.

G The preparation of an SQL application program includes precompilation, the preparation of its static SQL statements, and compilation of the modified source program. The exact steps required are product-specific.

© Copyright IBM Corp. 1982, 2002

Dynamic SQL

Programs containing embedded *dynamic* SQL statements must be precompiled like those containing static SQL, but unlike static SQL, the dynamic SQL statements are constructed and prepared at run time. The source form of a dynamic statement is a character string that is passed to the database manager by the program using the static SQL PREPARE and EXECUTE statements, or the EXECUTE IMMEDIATE statement. The operational form of the statement persists for the duration of the
G connection. For DB2 UDB for z/OS and OS/390 it only persists for the duration of
G the transaction.

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

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 workstation, prepares and executes them dynamically, and displays the results to the user. Such SQL statements are said to be issued *interactively*.

For example, the following facilities provide interactive capabilities:
- SPUFI for DB2 UDB for z/OS and OS/390
- Operations Navigator, Interactive SQL, or Query Manager for DB2 UDB for iSeries.
- Command Line Processor or Command Center for DB2 UDB for UWO.

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 to access SQL functions directly through a call interface. CLI programs can also be compiled using an Open Database Connectivity (ODBC) Software Developer's Kit, available from Microsoft or other vendors, enabling access to ODBC data sources. Unlike using embedded SQL, no precompilation is required. Applications developed using this interface 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.

For a complete description of all the available functions, see the product books.

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

DB2 provides 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.

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, see the product books.

Schemas

The objects in a relational database are organized into sets called schemas. A schema provides a logical classification of objects in the database. The schema-name is used as the qualifier of SQL object names such as tables, views, indexes, and triggers.

Each database manager supports a set of schemas that are reserved for use by the database manager. Such schemas are called *system schemas*. User objects must not be created in *system schemas*.

The schema SESSION and all schemas that start with 'SYS' and 'Q' are *system schemas*.

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 selects or generates, directly or indirectly, from one or more base tables. For more information about creating tables, see the "CREATE TABLE" on page 353.

Keys

A *key* is one or more columns that are identified as such in the description of an index, a unique constraint, or a referential constraint. The same column can be part of more than one key.

A *composite key* is an ordered set of two or more columns of the same base table. The ordering of the columns is not constrained by their ordering within the base 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 base table cannot have more than one primary key and the columns of the key must be defined as NOT NULL. A unique index on a primary key is called a primary index.

- Unique keys can be defined using the UNIQUE clause of the CREATE TABLE statement. A base table can have more than one set of UNIQUE keys. The columns of a UNIQUE key must be defined as NOT NULL.

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 base table is defined as a parent in a referential constraint, the default parent key is its primary key.

For more information on defining unique constraints, see"ALTER TABLE" on page 267 or "CREATE TABLE" on page 353.

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 base table containing the parent key is called the *parent table* of the referential constraint, and the base 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 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 base table that is a parent in at least one referential constraint. A base table can be defined as a parent in an arbitrary number of referential constraints.

Dependent table
A base table that is a dependent in at least one referential constraint. A base 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 base table is a descendent of base 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 base table in the set is a descendent of itself.

Self-referencing row A row that is a parent of itself.

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

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

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

The delete rule of a referential constraint applies when a row of the parent table is deleted. More precisely, the rule applies when a row of the parent table is the object of a delete or propagated delete operation (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.

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

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

- If D is a dependent of P and the delete rule is RESTRICT or NO ACTION, D is involved in the operation but is not affected by the operation.
- If D is a dependent of P and the delete rule is SET NULL, D is involved in the operation, and rows of D 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 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 base table that may be involved in a delete operation on P is said to be *delete-connected* to P. Thus, a base table is delete-connected to base table P if it is a dependent of P or a dependent of a base table to which delete operations from P cascade.

For more information on defining referential constraints, see“ALTER TABLE” on page 267 or “CREATE TABLE” on page 353.

Check Constraints

A *check constraint* is a rule that specifies which values are allowed in every row of a base table. The definition of a check constraint contains a search condition that must not be FALSE for any row of the base table. Each column referenced in the search condition of a check constraint on a table T, must identify a column of T. For more information on search conditions, see “Search Conditions” on page 126.

A base table can have more than one check constraint. Each check constraint defined on a base table is enforced by the database manager when:

- a row is inserted into that base table
- a row of that base table is updated.

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

For more information on defining check constraints, see “ALTER TABLE” on page 267 or “CREATE TABLE” on page 353.

Triggers

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

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

Triggers are 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 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 because the business rule is no longer repeated in several applications, but one version is centralized to the trigger. 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.

For more information about creating triggers, see the “CREATE TRIGGER” on page 368.

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 base 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 no search condition is specified or the specified 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.

- 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.

Indexes

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

Indexes are used by the database manager to:

- Improve performance. In most cases, access to data is faster than without an index.
- Ensure uniqueness. A base table with a unique index cannot have rows with identical keys.

An *index* is created with the CREATE INDEX statement. For more information about creating indexes, see the "CREATE INDEX" on page 338.

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.

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 constraint as the base table. Likewise, if the base table of a view is a parent table, DELETE operations using that view are subject to the same rules as DELETE operations on the base table.

A *view* is created with the CREATE VIEW statement. For more information about creating views, see the "CREATE VIEW" on page 376.

Aliases

An *alias* is an alternate name for a table or view. An alias can be used to reference a table or view in cases where an existing table or view can be referenced. However, the option of referencing a table or view by an alias is not explicitly shown in the syntax diagrams or mentioned in the description of SQL statements. Like tables and views, an alias may be created, dropped, and have a comment 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* is created with the CREATE ALIAS statement. For more information about creating aliases, see the "CREATE ALIAS" on page 303.

Packages and Access Plans

A *package* is an object that contains control structures used to execute SQL statements. Packages are produced during program preparation. The control structures can be thought of as the bound or operational form of SQL statements. 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 other product-specific objects 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 395). This means that any packages, procedures, functions, triggers, and any product-specific objects containing control structures referencing the dropped object are invalidated.

In some cases, an invalidated *access plan* may 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.

Routines

A *routine* is an executable SQL object. There are two types of routines.

Functions

A *function* is a routine that can be invoked from within other SQL statements and returns a value. For more information, see "Functions" on page 91.

A *function* is created with the CREATE FUNCTION statement. For more information about creating functions, see the "CREATE FUNCTION" on page 310.

Procedures

A *procedure* (sometimes called a *stored procedure*) is a routine that can be called to perform operations that can include both 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 which may or may not contain SQL statements.

A *procedure* is created with the CREATE PROCEDURE statement. For more information about creating procedures, see the "CREATE PROCEDURE" on page 340.

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. Host languages can easily call procedures that exist on the local system. SQL can also easily 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. There are two ways this can be done. Without procedures, when the first SQL statement is executed, the application requester will send a request to an application server to

perform the operation. It then waits 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 an application 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 then waits for a single reply that indicates whether the CALL statement executed successfully or not and optionally returns results.

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

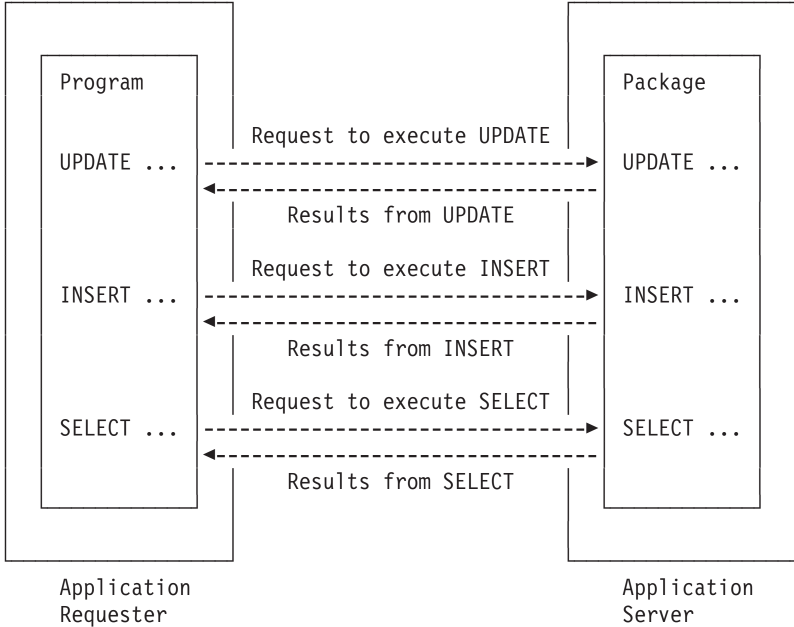


Figure 1. Application Without Remote Procedure

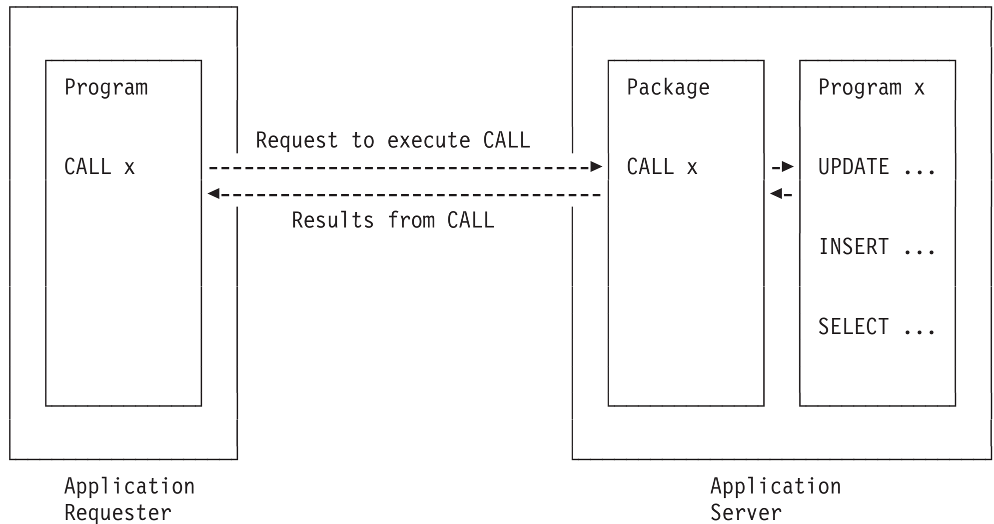


Figure 2. Application With Remote Procedure

Catalog

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

Tables and views in the catalog are smilar to any other database tables and views. Any user that has the SELECT privilege, can read the data in the catalog tables and views. A user cannot directly modify a catalog table or view, however. The database manager ensures that the catalog contains accurate descriptions of the objects in the database at all times.

G For further information about the catalog, see the product books.

Application Processes, Concurrency, and Recovery

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

G
G

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

G The locking facilities of the database managers are similar but not identical. One of the common properties is that each of the database managers can acquire locks in order to prevent uncommitted changes made by one application process from being perceived by any other. The database manager will release all locks it has

acquired on behalf of an application process when that process ends, but an application process itself can also explicitly request that locks be released sooner. This operation is called *commit*.

G
G

Like the locking facilities, the recovery facilities of the database managers are similar but not identical. One common property is that each of the database managers provides a means of *backing out* uncommitted changes made by an application process. This might be necessary in the event of a failure on the part of an application process, or in a *deadlock* situation. An application process itself, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

A *unit of work* (also called a *transaction*, logical unit of work, or unit of recovery) is a recoverable sequence of operations within an application process. At any time, an application process has at most a single unit of work, but the life of an application process may involve many units of work as a result of commit or rollback operations.

A unit of work is started when an application process is started. A unit of work is ended by a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it ends. While these changes remain uncommitted, other application processes are unable to perceive them and they can be backed out. [3] Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback.

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

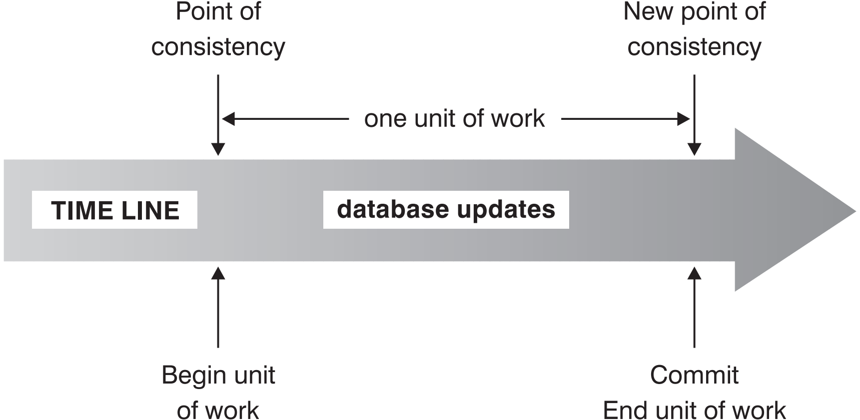


Figure 3. Unit of Work with a Commit Operation

3. Except for isolation level uncommitted read, described in “Uncommitted Read (UR)” on page 15.

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

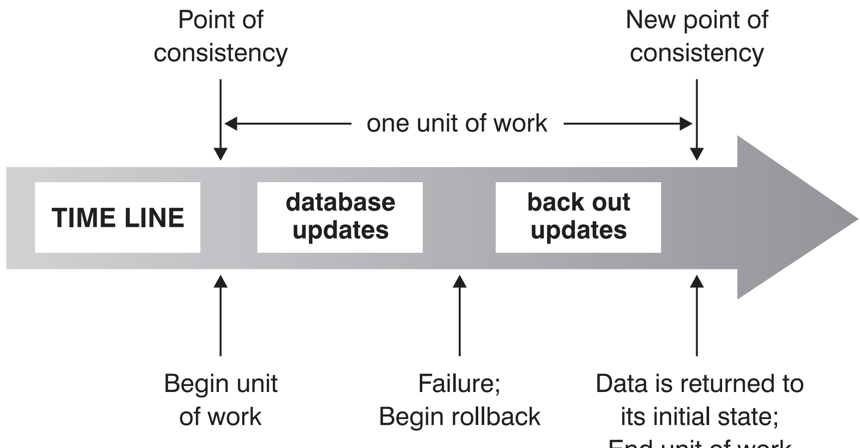


Figure 4. Rolling Back Changes from a Unit of Work

G **Note:** In addition to relational databases, the environment in which an SQL
G program executes may also include other types of recoverable resources. If
G this is the case, the scope and acceptability of the SQL COMMIT and
G ROLLBACK statements depend on the environment.

G In DB2 UDB for z/OS and OS/390 and DB2 UDB for iSeries, a lock that protects
G the current row of a cursor from updates or deletes by concurrent application
G processes also protects the row from Positioned UPDATEs and Positioned
G DELETEs that reference another cursor of the same application process. [4] In DB2
G UDB for UWO this protection does not apply.

Isolation Level

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

- The degree to which rows retrieved by P are available to other concurrently executing application processes.
- The degree to which database changes made by concurrently executing application processes 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*.

G Each product provides a product-specific means of explicitly specifying a default isolation level:

- For static SQL statements, the *default isolation level* is the isolation level specified when the containing package, procedure, function, or trigger was created.

4. In DB2 UDB for iSeries, Searched UPDATEs and Searched DELETEs are also included.

- For dynamic SQL statements, the *default isolation level* is isolation level specified for the application process.

Products support these isolation levels by automatically locking the appropriate data. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. Each database manager supports at least two types of locks:

Share Limits concurrent application processes to read-only operations on the data.

Exclusive Prevents concurrent application processes from accessing the data in any way except for application processes with an isolation level of *uncommitted read*, which can read but not modify the data. (See "Uncommitted Read (UR)" on page 15.)

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 application process is guaranteed at least the minimum requested lock level.

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

Repeatable Read (RR)

Level RR ensures that:

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

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

In the SQL 1999 Core standard, 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 application processes until the unit of work is complete. [5]
- Any row changed by another application process cannot be read until it is committed by that application process.

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

---

5. 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.

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

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

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

In the SQL 1999 Core standard, Read Stability is called Repeatable Read.

Cursor Stability (CS)

Like levels RR and RS, level CS ensures that any row changed by another application process cannot be read until it is committed by that application process. Unlike RR and RS, level CS only ensures that the current row of every updatable cursor is not changed by other application processes. Thus, the rows read during a unit of work can be changed by other application processes. In addition to any exclusive locks, an application process running at level CS has at least a share lock for the current row of every one of its open cursors.

In the SQL 1999 Core standard, Cursor Stability is called Read Committed.

Uncommitted Read (UR)

For a SELECT INTO, 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 application processes.
- Any row changed by another application process to be read even if the change has not been committed by that application process.

G G G For other operations, the rules of level CS apply. In DB2 UDB for z/OS and OS/390, UR is escalated to CS for a subquery used in a DELETE or UPDATE statement, or for a subselect used in an INSERT statement.

In the SQL 1999 Core standard, Uncommitted Read is called Read Uncommitted.

Comparison of Isolation Levels

The following table summarizes information about isolation levels.

	UR	CS	RS	RR
Can the application see uncommitted changes made by other application processes?	Yes	No	No	No
Can the application update uncommitted changes made by other application processes?	No	No	No	No
Can the re-execution of a statement be affected by other application processes? *See phenomenon P3 (phantom) below.*	Yes	Yes	Yes	No
Can "updated" rows be updated by other application processes?	No	No	No	No

Can "updated" rows be read by other application processes that are running at an isolation level other than UR?	No	No	No	No
Can "updated" rows be read by other application processes that are running at the UR isolation level?	Yes	Yes	Yes	Yes
Can "accessed" rows be updated by other application processes? For RS, "accessed rows" typically means rows selected. For RR, see the product-specific documentation. *See phenomenon P2 (nonrepeatable read) below*.	Yes	Yes	No	No
Can "accessed" rows be read by other application processes?	Yes	Yes	Yes	Yes
Can "current" row be updated or deleted by other application processes? *See phenomenon P1 (dirty-read) 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, 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. Each computer system has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given database manager to execute SQL statements on another computer system.

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

these considerations and can operate as if it were accessing a local database. A simple distributed relational database environment is illustrated in Figure 5.

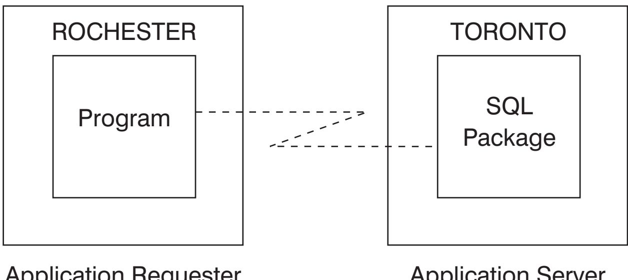


Figure 5. A Distributed Relational Database Environment

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

Application Servers

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

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

An application server can be local to, or remote from, the environment where the process is started. (An application server is present, even when not using distributed relational databases.) This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement.
G The format and maintenance of this directory are product-specific.

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

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 application server to which it is currently connected, even though that application might be running via the application requester of a database manager that does not support some of those statements and clauses. Restrictions are listed in "Considerations for Using Distributed Relational Database" on page 520.

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 296.
- CONNECT (Type 2) is used for application-directed distributed unit of work. See “CONNECT (Type 2)” on page 300.

See Appendix G, “CONNECT (Type 1) and CONNECT (Type 2) Differences” on page 591 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 application process at computer system A can connect to an application server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the application process can connect to an application server at computer system C, and so on.

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

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

Remote Unit of Work Connection Management

An application process 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:

Begin process

CONNECT

Connectable and Connected

Successful CONNECT

Connectable and Unconnected

CONNECT with system failure or COMMIT with the connection in the release-pending state.

SQL statement other than CONNECT, COMMIT, or ROLLBACK

ROLLBACK or successful COMMIT

System failure with rollback

Unconnectable and Connected

Figure 6. Remote Unit of Work Application Process Connection State Transitions

The initial state of an application process is *connectable* and *connected*. The application server to which the application process is connected is determined by a product-specific option that 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 application process cannot be implicitly connected to an application server more than once. The other rules for implicit CONNECT operations are product-specific.

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

The connectable and unconnected state: An application process is not connected to an application server. The only SQL statement that can be executed is CONNECT.

The application process enters this state when:

- The connection was in a connectable state, but the CONNECT statement was unsuccessful.
- The connection was in a release-pending state, and a COMMIT operation is performed.

G The other reasons for entering this state are product-specific.

G In DB2 UDB for z/OS and OS/390, an application process can also be in the
G *unconnectable* and *unconnected* state. An application process enters this state as a
G result of a system failure that has caused a rollback at the application server. An
G application process in this state must execute a rollback operation.

Consecutive CONNECT statements can be executed successfully because CONNECT does not remove the application process from the connectable state. A CONNECT to the application server to which the application process 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, RELEASE, ROLLBACK, or SET CONNECTION. 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* [6] *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 application process at computer system A can connect to an application 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 application server. However, unlike remote unit of work, any number of application servers can participate in the same unit of work. A commit or rollback operation ends the unit of work.

Application-Directed Distributed Unit of Work Connection Management

At any time:

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

Initial state of an application process: An application process is initially in the connected state and has exactly one connection. The initial states of a connection are current and held.

The following diagram shows the state transitions:

6. For DB2 UDB for z/OS and OS/390, the term used is *DRDA access* where the application issues explicit CONNECT statements.

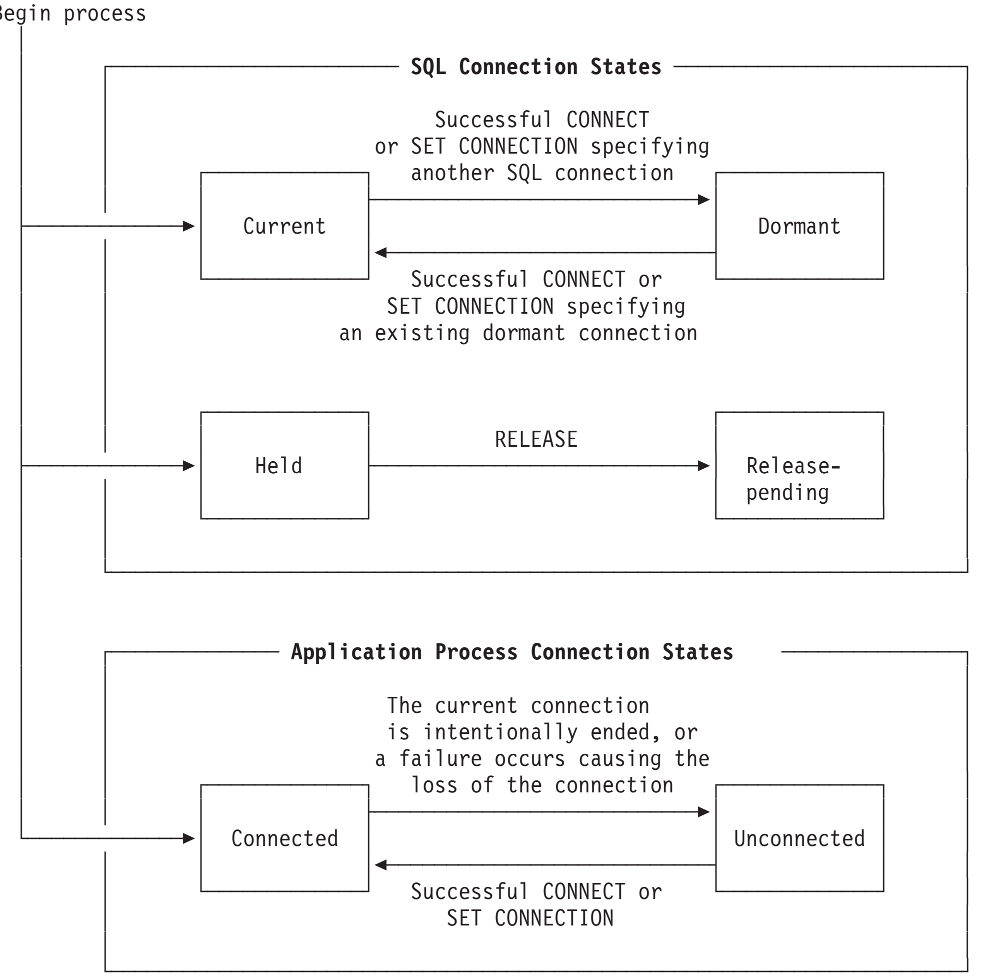


Figure 7. Application-Directed Distributed Unit of Work Connection and Application Process Connection State Transitions

Connection States

If an application process successfully executes a CONNECT statement:

- The current connection is placed in the dormant and held state, and
- 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 application process, an error is returned.

A connection in the dormant state is placed in the current state using the SET CONNECTION statement. [7] 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 application process 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.

7. Some products provide a product-specific option that allows the CONNECT statement to place a connection in the dormant state.

A connection is placed in the release-pending state by the RELEASE statement. When an application process executes a commit operation, every release-pending connection of the process is ended. Changing the state of 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.

Application Process Connection States

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

- An application process cannot have more than one connection to the same application server at the same time.
- When an application process executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the application process.
- When an application process executes a CONNECT statement, the specified server name must not be an existing connection in the set of connections of the application process. [8]

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

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

If an application process does not have a current connection, the application process 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, SET CONNECTION, RELEASE, COMMIT, and ROLLBACK.

An application process in the connected state enters the unconnected state when its current connection is intentionally ended or the execution of an SQL statement is unsuccessful because of a failure that causes a rollback operation at the current server and loss of the connection. Connections are intentionally ended when an application process successfully executes a commit operation and the connection is in the release-pending state.

When a Connection is Ended

When a connection is ended, all resources that were acquired by the application process through the connection and all resources that were used to create and maintain the connection are deallocated. For example, if application process P has placed the connection to application 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 application process is placed in the unconnected state. All connections of an application process are ended when the application process ends.

8. In DB2 UDB for z/OS and OS/390, this rule is enforced only if the SQLRULES(STD) bind option is specified.

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 of the data and the environment type of the sending system. 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 DB2 UDB for z/OS and OS/390 application server, DB2 UDB for z/OS and OS/390, knowing the data type and the sending system, converts the number from IEEE format to S/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 that is to be performed with that data. Strings are converted in accordance with the IBM Character Data Representation Architecture (CDRA). For more information on character conversion, refer to *Character Data Representation Architecture Level 1 Reference*, SC09-2190.

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*. [9]

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.

Note that SQL statements are character strings and are therefore subject to character conversion.

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:
  - 26 nonaccented letters A through Z
  - 26 nonaccented letters a through z
  - digits 0 through 9

9. 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.

- . , : ; ? ( ) ' " / − _ & + % * = < >

code page
A set of assignments of characters to code points. In EBCDIC, for example, "A" is assigned code point X'C1' and "B" is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.

code point
A unique bit pattern that represents a character.

coded character set
A set of unambiguous rules that establishes a character set and the one-to-one relationships between the characters of the set and their coded representations.

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

- Single-byte EBCDIC
- Single-byte ASCII [10]
- Double-byte EBCDIC
- Mixed single- and double-byte ASCII
- Unicode (UTF-8, UTF-16, and UCS-2 universal coded character sets).

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.

10. The term ASCII is used throughout this book to refer to several encodings such as IBM-PC, ISO 8, or ISO 7 data.

code page: pp1 (ASCII)

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

code point: 2F

character set ss1 (in code page pp1)

code page: pp2 (EBCDIC)

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

character set ss1 (in code page pp2)

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

A CCSID of a native encoding scheme identifies one of the coded character sets in which data may be stored at that site. A CCSID of a foreign encoding scheme identifies 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 coded character set with an EBCDIC or Unicode encoding scheme, but not in an ASCII encoding scheme.

G

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. The rules are product-specific.

For details on character conversion, see:

- "Conversion Rules for Assignments" on page 61
- "Conversion Rules for Comparison" on page 66
- "Conversion Rules for Operations that Combine Strings" on page 71
- "Considerations for Using Distributed Relational Database" on page 520.

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 a length is an attribute of strings. All values of the same string column have the same CCSID. In DB2 UDB for UWO, support for CCSIDs is limited to DRDA. CCSIDs are mapped into code page identifiers when receiving DRDA flows and code page identifiers are mapped into CCSIDs when sending DRDA flows.[11]

In each database manager, character conversion involves the use of a *CCSID Conversion Selection Table*. [12] 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.)

Different types of conversions may be supported by each database manager. Round-trip conversions attempt to preserve characters in one CCSID that are not defined in the target CCSID so that if the data is subsequently converted back to the original CCSID, the same original characters result. Enforced subset match conversions do not attempt to preserve such characters. Which type of conversion is used for a specific source and target CCSID is product-specific. For more information, see IBM's Character Data Representation Architecture (CDRA).

Default CCSID

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

- String constants (including string constants that represent datetime values)
- Special registers with string values (such as USER and CURRENT SERVER)
- CAST specifications where the result is a character or graphic string
- The result of the CHAR, CLOB, DBCLOB, DIGITS, HEX, GRAPHIC, SPACE, VARCHAR, and VARGRAPHIC scalar functions
- String columns defined by CREATE TABLE and ALTER TABLE statements.

In a distributed application, the default CCSID of host variables is determined by the application requester. In a nondistributed application, the default CCSID of host variables is determined by the application server.

---

11. DB2 UDB for UWO operating outside DRDA does support conversions between the client and the server based on code page.

12. In some implementations, the Conversion Selection Table is part of the operating system and is accessed indirectly by the database manager. In other implementations, it is a catalog table, or a combination of both.

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.

  For further information about administrative authority, see the product references.

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).

  - In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO, the set consists of all users (including future users), including those with privately granted privileges on the table or view.
  - In DB2 UDB for iSeries, 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*.

  - In DB2 UDB for z/OS and OS/390 a revoke of a privilege from an authorization ID only revokes the privilege granted by a specific authorization ID. For example, assume that the SELECT has been granted to CHRIS by CLAIRE and also by BOBBY. If CLAIRE revokes the SELECT privilege from CHRIS, CHRIS still has the SELECT privilege that was granted by BOBBY.

    Revoking a privilege from an authorization ID will also revoke that same privilege from all other authorization IDs that were granted the privilege by that authorization ID. For example, assume CLAIRE grants SELECT WITH GRANT OPTION to RICK, and RICK then grants SELECT to BOBBY and CHRIS. If CLAIRE revokes the SELECT privilege from RICK, the SELECT privilege is also revoked from both BOBBY and CHRIS.
  - In DB2 UDB for iSeries, and DB2 UDB for UWO, a revoke of a privilege from an authorization ID revokes the privilege granted by all authorization IDs.

G G G 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
- G it has a product-specific privilege.

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership means the user is authorized to reference the object in any SQL statement. The privileges on the object can be granted by the owner, and cannot be revoked from the owner.

When an object is created, the authorization ID of the statement is the *owner* of an object if:

- the object name in the CREATE statement is not qualified, or
- the explicitly specified schema name is the same as the authorization ID of the statement.

G G Otherwise, the owner of the object is product-specific and the privileges held by the authorization ID of the statement must include administrative authority.

Storage Structures

G G G *Storage structures* (spaces for tables and indexes for example) differ between each DB2 relational database product. For detailed information about storage structures, see the product references.

Chapter 2. Language Elements

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

Characters

The basic symbols of keywords and operators in the SQL language are single-byte characters 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.

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

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

- space
- " quotation mark or double-quote
- % percent
- & ampersand
- ' apostrophe or single quote
- (left parenthesis
-) right parenthesis
- * asterisk
- + plus sign
- , comma
- | vertical bar[13]
- { left brace
- } right brace
- – minus sign
- . period
- / slash
- : colon
- ; semicolon
- < less than
- = equals
- > greater than
- ? question mark
- _ underline or underscore
- [left bracket
-] right bracket

13. Using the vertical bar (|) character might inhibit code portability between IBM relational database products. Use the CONCAT operator in place of the || operator.

© Copyright IBM Corp. 1982, 2002

Tokens

The basic syntactic units of the language are called *tokens*. A token consists of one or more characters, excluding the blank character 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.

  Examples

  ```
  1       .1       +2       SELECT       E       3
  ```

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark (?) is also a delimiter token when it serves as a parameter marker, as explained under "PREPARE" on page 433.

  Examples

  ```
  ,     'Myst Island'       "fld1"       =       .
  ```

Spaces: A *space* is a sequence of one or more blank characters. Tokens, other than string constants and delimited identifiers, must not include a space. Any token can be followed by a space. Every ordinary token must be followed by a delimiter token or a space. If the syntax does not allow an ordinary token to be followed by a delimiter token, that ordinary token must be followed by a space.

Comments: Static SQL statements may include host language comments or SQL comments. For more information on host language comments see the Host Language Appendices. Either type of comment may be specified wherever a space may be specified, except within a delimiter token or between the keywords EXEC and SQL. In Java, SQL comments are not allowed within embedded Java expressions. See Appendix J, "Coding SQL Statements in Java Applications" on page 627. SQL comments are introduced by two consecutive hyphens (--) and ended by the end of the line. For more information, see "SQL Comments" on page 266.[14]

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

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

is equivalent, after folding, to:

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

14. In DB2 UDB for z/OS and OS/390, the precompiler option STDSQL(YES) must be used to allow SQL comments.

Identifiers

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

SQL Identifiers

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

- An *ordinary identifier* is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier should not be a reserved word. See “Reserved Words” on page 691 for a list of reserved words. If a reserved word is used as an identifier in SQL, it must be specified in uppercase and must be a delimited identifier or specified in a host variable.
- A *delimited identifier* is a sequence of characters enclosed within quotation marks ("). The sequence must consist of one or more characters of the SQL language. 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 quotation marks.

Examples

```
WKLYSAL    WKLY_SAL    "WKLY_SAL"    "UNION"    "wkly_sal"
```

See Table 1 on page 37 for information on the maximum length of identifiers.

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. Names beginning with 'DB2', 'SQ'[15], 'SQL', 'sql', 'RDI', or 'DSN' should not be used because precompilers generate host variable names that begin with these characters.

See Table 1 on page 37 for the limits on the maximum size of the host identifier name imposed by each product.

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

Naming Conventions

The rules for forming a name depend on the type of the object designated by the name. Many database objects have a *schema qualified name*. A *schema qualified name* may consist of a single SQL identifier (in which case the *schema-name* is implicit) or a *schema-name* followed by a period and an SQL identifier. The following list defines these terms.

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

authorization-name
An SQL identifier that designates a user or group of users. An *authorization-name* must not be a delimited identifier that includes lowercase letters or special characters. See "Authorization IDs and Authorization Names" on page 41 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 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.

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

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

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

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

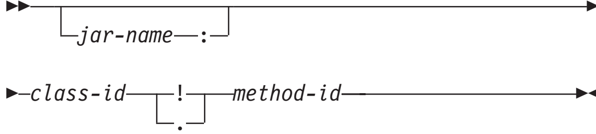
descriptor-name
A host variable name that designates an SQL descriptor area (SQLDA). See "References to Host Variables" on page 85 for a description of a host variable. Note that *descriptor-name* never includes an indicator variable.

distinct-type-name
A qualified or unqualified name that designates a distinct type. The unqualified form of a *distinct-type-name* is an SQL identifier. An unqualified *distinct-type-name* in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the *distinct-type-name* appears as described by the rules in "Unqualified Distinct Type,

Function, Procedure, and Specific Names” on page 38. The qualified form is a *schema-name* followed by a period and an SQL identifier.

external-program-name
There are two distinct forms of an *external-program-name* that designate an external program.

- In C and COBOL, *external-program-name* is an SQL identifier.
- In Java, *external-program-name* is a character string. The format of the character string is an optional jar-name, 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’).



jar-name
A case-sensitive string that designates a JAR.

class-id
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 myPackage/ directory for the StoredProcs classes.

For details regarding the location or installation of Java classes, see the product documentation.

method-id
The method-id identifies the method name of the public, static Java method to be invoked.

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

function-name
A qualified or unqualified name that designates a function. The unqualified form of a *function-name* is an SQL identifier. An unqualified function-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the function appears as described by the rules in “Unqualified Distinct Type, Function, Procedure, and Specific

Names" on page 38. The qualified form is a *schema-name* followed by a period and an SQL identifier.

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 85.

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

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

package-name
A qualified or unqualified name that designates a package. The unqualified form of a *package-name* is an SQL identifier. A *package-name* must not be a delimited identifier that includes lowercase letters or special characters. An unqualified package-name in an SQL statement is implicitly qualified by the default schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

G G G In DB2 UDB for z/OS and OS/390, a package-name in an SQL statement must be qualified.

parameter-name
An SQL identifier that designates a parameter in an SQL procedure or SQL function. A *parameter-name* must not be a delimited identifier that includes lowercase letters or special characters.

procedure-name
A qualified or unqualified name that designates a procedure. The unqualified form of a *procedure-name* is an SQL identifier. The implicit qualifier is a schema name, which is determined by the context in which the function appears as described by the rules in "Unqualified Distinct Type, Function, Procedure, and Specific Names" on page 38. The qualified form is a *schema-name* followed by a period and an SQL identifier.

schema-name
An SQL identifier that provides a logical grouping for SQL objects. A *schema-name* is used as the qualifier of the name of SQL objects (see "Reserved Schema Names" on page 691).

G G In DB2 UDB for iSeries, a blank is not allowed in a delimited schema name.

server-name
An SQL identifier that designates an application

server. The identifier must start with a letter and must not include lowercase letters or special characters.

specific-name
A qualified or unqualified name that designates a function or procedure. The unqualified form of a *specific-name* is an SQL identifier. The implicit qualifier is a schema name, which is determined by the context in which the specific name appears as described by the rules in "Unqualified Distinct Type, Function, Procedure, and Specific Names" on page 38. The qualified form is a *schema-name* followed by a period and an SQL identifier.

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

SQL-parameter-name
A qualified or unqualified name that designates a parameter in the SQL routine body of an SQL procedure or SQL function. The unqualified form of an SQL parameter name is an SQL identifier. An *SQL-parameter-name* must not be a delimited identifier that includes lowercase letters or special characters. The qualified form is a *function-name* or *procedure-name* followed by a period and an SQL identifier.

SQL-variable-name
A qualified or unqualified name that designates a variable in the SQL routine body of an SQL procedure. The unqualified form of an SQL variable name is an SQL identifier. An *SQL-variable-name* must not be a delimited identifier that includes lowercase letters or special characters. The qualified form is an SQL label followed by a period and an SQL identifier.

statement-name
An SQL identifier that designates a prepared SQL statement.

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

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

view-name
A qualified or unqualified name that designates a view. The unqualified form of a *view-name* is an SQL identifier. An unqualified view-name in an SQL statement is implicitly qualified by the default

schema. The qualified form is a *schema-name* followed by a period and an SQL identifier.

Table 1. Identifier Length Limits (in bytes)

Identifier Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Longest authorization name	8	10	30	8
Longest condition name	64	128	64	64
Longest constraint name	8[82]	128	18	8[82]
Longest correlation name	18	128	128	18
Longest cursor name	18	18	18	18
Longest external program name (unqualified form)	8	10	18	8
Longest external program name (string form)	1305	279	254	254
Longest host identifier[79]	64	64	255	64
Longest schema name	8[78]	10	8[80]	8
Longest server name	16	18	8	18
Longest statement name	18	18	18	18
Longest SQL label	64	128	64	64
Longest unqualified alias name	18	128	128	18
Longest unqualified column name	18	30	30	18
Longest unqualified distinct type name	18	128	18	18
Longest unqualified function name	18	128	18	18
Longest unqualified index name	18	128	18	18
Longest unqualified package name	8	10	8	8
Longest unqualified procedure name	18	128	128	18
Longest unqualified specific name	18	128	128	18
Longest unqualified SQL parameter name	18	128	64[81]	18
Longest unqualified SQL variable name	18	128	64	18
Longest unqualified table and view name	18	128	128	18
Longest unqualified trigger name	8	128	18	18

SQL Path

The *SQL path* is an ordered list of schema names. The database manager uses the SQL path to resolve the schema name for unqualified data 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 SQL path from left to right, the database manager implicitly qualifies the object name with the first schema name in the SQL path that contains the same object with the same unqualified name. For 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 and number of parameters but different parameter data types may be defined in the same schema or other schemas on the SQL path. For details, see “Function Resolution” on page 92.

For example in DB2 UDB for iSeries, 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 SQL path used depends on the SQL statement:

- For static SQL statements, the *SQL path* used is the SQL path specified when the
G containing package, procedure, function, trigger, or view was created. The way
G the SQL path is specified is product-specific.
- For dynamic SQL statements, the *SQL path* is the value of the CURRENT PATH special register. CURRENT PATH can be set by the SET PATH statement. For more information, see “SET PATH” on page 461.

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” on page 76.

Qualification of Unqualified Object Names

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

Unqualified Alias, Index, Package, Table, Trigger, and View Names

Unqualified alias, index, package, table, trigger, and view names are implicitly
G qualified by the *default schema*. Each product provides a product-specific means of explicitly specifying a default schema:

- For static SQL statements, the *default schema* is the default schema specified when the containing function, package, procedure, or trigger was created.
- For dynamic SQL statements, the *default schema* is the default schema specified for the application process.

If the default schema is not explicitly specified, the default schema is the authorization ID of the statement.

Unqualified Distinct Type, Function, Procedure, and Specific 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, Index, Package, Table, Trigger, and View Names" on page 38).
- 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 SQL 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 SQL path such that the schema contains a procedure with the same name and the same number of parameters.
  – For function names, the database manager uses the SQL path in conjunction with function resolution, as described under "Function Resolution" on page 92.
  – For specific names specified for sourced functions, see "CREATE FUNCTION (Sourced)" on page 325.

Aliases

An *alias* can be thought of as an alternative name for a table or view. 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 or view within the same relational database.

An alias can be used wherever a table or view name can be used. However, 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 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 an alias is created. An alias cannot refer to another alias. An alias can only refer to a table or view within the same relational database.

The option of referring to a table or view 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, 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 at statement preparation time by the qualified base table or view 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 recreating the alias has no effect on the view or index.

Authorization IDs and Authorization Names

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

Authorization IDs 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 applies to a static SQL statement is the authorization ID that is used during program preparation. The authorization ID that applies to a dynamic SQL statement is the authorization ID that was obtained by the database manager when a connection was established between the database manager and the process.[16] This is called the *run-time authorization ID*.

An *authorization-name* specified in an SQL statement should not be confused with the authorization ID of the statement. An *authorization-name* is an identifier that is used 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.

Example

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

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

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

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

16. In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO the DYNAMICRULES bind option setting can impact the authorization ID that applies to a dynamic SQL statement. For details, refer to product specific documentation.

Data Types

The smallest unit of data that can be manipulated in SQL is called a *value*. How values are interpreted depends on the *attributes* of their source, which includes the data type, length, precision, scale and CCSID. The sources of values are:

Columns

Constants

Expressions

Functions

Special registers.

Variables (such as host variables, SQL variables, parameter markers and parameters of routines)

The 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 51.

Figure 8 illustrates the various data types supported by the DB2 UDB relational database products:

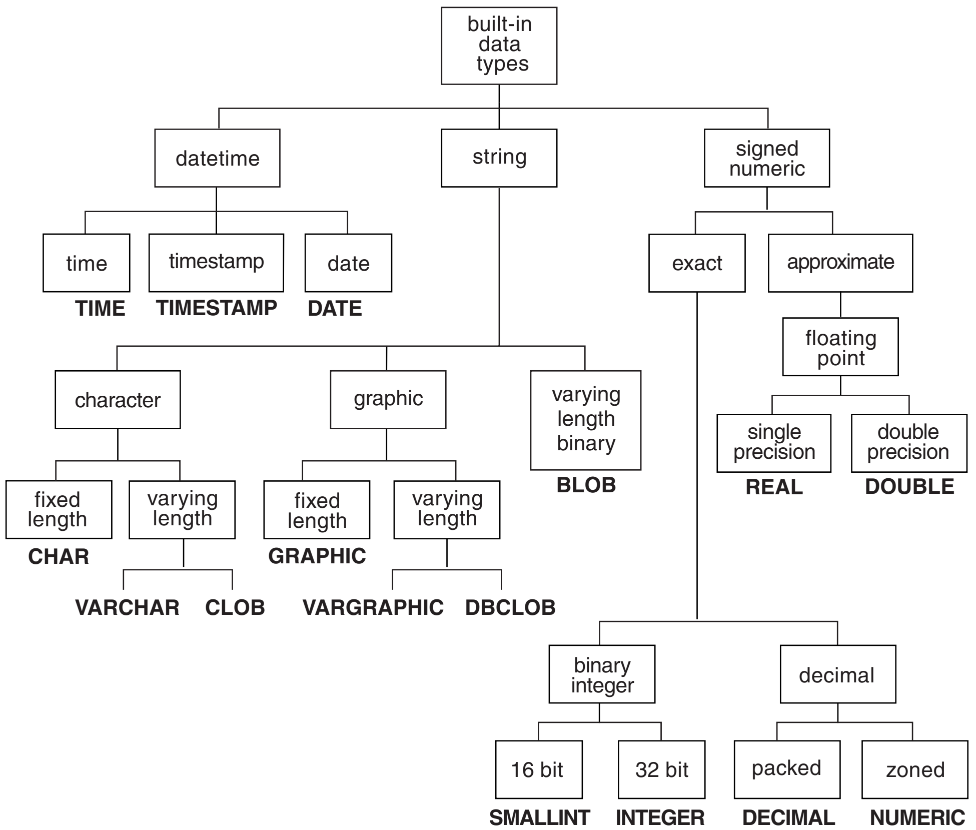


Figure 8. Data Types Supported by the DB2 UDB Relational Database Products

G G G G In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, zoned decimal is not supported as a native data type and NUMERIC is treated as a synonym for DECIMAL. Zoned decimal numbers received through DRDA protocols are converted to packed decimal.

Nulls

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

Numbers

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

All numbers have a *sign*, a *precision*, and a *scale*. If a column value is zero, the sign is positive. The precision is the total number of binary or decimal digits excluding the sign. The scale is the total number of binary or decimal digits to the right of the decimal point. If there is no decimal point, the scale is zero.

Small Integer

A *small integer* is a binary number composed of 2 bytes with a precision of 5 digits and a scale of zero. The range of small integers is −32 768 to +32 767.

Large Integer

A *large integer* is a binary number composed of 4 bytes with a precision of 10 digits and a scale of zero. The range of large integers is −2 147 483 648 to +2 147 483 647.

Floating-Point

A *single-precision floating-point* number is a 32-bit approximate representation of a real number. The number can be zero or can range from $-3.4x10^{38}$ to $-1.17x10^{-37}$, or from $+1.17x10^{-37}$ to $+3.4x10^{38}$.

A *double-precision floating-point* number is a 64-bit approximate representation of a real number. The number can be zero or can range from $-7.2x10^{75}$ to $-5.4x10^{-79}$, or from $+5.4x10^{-79}$ to $+7.2x10^{75}$.

See Table 39 on page 510 for more information.

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.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is *-n* to *+n*, where the absolute value of *n* is the largest number that can be represented with the applicable precision and scale.

The maximum range is $-10^{31} + 1$ to $10^{31} - 1$.

Numeric Host Variables

Binary integer variables can be declared in all host languages. Decimal variables can be declared in all host languages except C.

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
G is used to interpret the string. The mechanism to specify the default decimal
G separator character is product-specific.

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

When fixed-length character string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a CHAR column, the length attribute must be between 1 and 254 inclusive. See Table 39 on page 510 for more information.

Varying-Length Character Strings

The types of varying-length character strings are:

- VARCHAR
- CLOB

A *Character Large OBject* (CLOB) column is useful for storing large amounts of character data, such as documents written using a single character set.

Distinct types, columns, and variables all have length attributes. When varying-length character string distinct types, columns, and variables are defined, the maximum length is specified and this becomes the length attribute. Actual values may have a smaller length. For a VARCHAR column, the length attribute must be between 1 and 32 672 inclusive. For a CLOB column, the length attribute must be between 1 and 2 147 483 647 inclusive. See Table 39 on page 510 for more information.

For the restrictions that apply to the use of VARCHAR strings longer than 255, see "Limitations on Use of Strings" on page 48.

Character-String Variables

- Fixed-length character-string variables can be declared in all host languages except REXX and Java. (In C, fixed-length character string variables are limited to a length of 1.)
- Varying-length character-string variables can be used in all host languages except CLOBs cannot be used in REXX.

For information on how to code in a host language, refer to the host language appendices.

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 therefore never converted. The CCSID for bit data is X'FFFF'
G (65535). In DB2 UDB for UWO, the CCSID for bit data is X'0000'
G (zero).

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

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

Unicode data Data that contains characters represented by one or more bytes. Each Unicode character string is encoded using UTF-8. Each Unicode string has an associated CCSID.

In DB2 UDB for UWO, support for CCSIDs is limited to DRDA. CCSIDs are mapped into code page identifiers when receiving DRDA flows and code page identifiers are mapped into CCSIDs when sending DRDA flows.

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

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

In DB2 UDB for UWO, ASCII is the native encoding scheme, and in DB2 UDB for z/OS and OS/390, and DB2 UDB for iSeries, EBCDIC is the native encoding scheme. DB2 UDB for z/OS and OS/390 supports ASCII encoded data. Because of the shift characters, EBCDIC mixed data requires more storage than ASCII mixed data.

Examples

元gen 気ki needs CHAR(9) in ASCII.

SO元SI gen SO気SI ki needs CHAR(13) in EBCDIC.

To minimize the effects of these differences, use varying-length strings with an appropriate declared length in applications that require mixed data and operate on both ASCII and EBCDIC systems.

Graphic Strings

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

Fixed-Length Graphic Strings

When fixed-length graphic string distinct types, columns, and variables are defined, the length attribute is specified and all values have the same length. For a GRAPHIC column, the length attribute must be between 1 and 127 inclusive. See Table 39 on page 510 for more information.

Varying-Length Graphic Strings

The types of varying-length graphic strings are:

- VARGRAPHIC
- DBCLOB

A *Double-Byte Character Large OBject* (DBCLOB) column is useful for storing large amounts of double-byte character data, such as documents written using a double-byte character set.

Distinct types, columns, and variables all have length attributes. When varying-length graphic string distinct types, columns, and variables are defined, the maximum length is specified and this becomes the length attribute. Actual values may have a smaller length. For a VARGRAPHIC column, the length attribute must be between 1 and 16 336 inclusive. For a DBCLOB column, the length attribute must be between 1 and 1 073 741 823 inclusive. See Table 39 on page 510 for more information.

For the restrictions that apply to the use of VARGRAPHIC strings longer than 127, see "Limitations on Use of Strings" on page 48.

Graphic-String Variables

- Fixed-length graphic-string variables can be declared in all host languages except REXX and Java. (In C, fixed-length graphic-string variables are limited to a length of 1.)
- Varying-length graphic-string variables can be declared in all host languages except DBCLOBs cannot be used in REXX.

For information on how to code in a host language, refer to the host language appendices.

Graphic Encoding Schemes

Each graphic string is further defined as one of:

DBCS data
Data in which every character is represented by a character from the double-byte character set (DBCS). 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.

Unicode data
Data that contains characters represented by two or four bytes. Each Unicode graphic string is encoded using either UCS-2 or UTF-16. Each Unicode string has an associated 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 Large OBject* (BLOB) column is useful for storing large amounts of non-character data, such as pictures, voice and mixed media. Another use is to hold structured data for exploitation by distinct types and user-defined functions.

Distinct types, columns, and variables all have length attributes. When varying length distinct types, columns, and variables are defined, the maximum length is specified and this becomes the length attribute. Actual values may have a smaller length. For a BLOB column, the length attribute must be between 1 and 2 147 483 647 bytes inclusive. See Table 39 on page 510 for more information.

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

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 to change a FOR BIT DATA character string into a BLOB string.

Large Objects (LOBs)

The term *large object* and the generic acronym *LOB* are used to refer to any CLOB, DBCLOB, or BLOB data type.

Manipulating Large Objects (LOBs) with Locators

Since LOB values can be very large, the transfer of these values from the database server to client application program host variables can be time consuming. Also, application programs typically process LOB values a piece at a time, rather than as a whole. For these cases, the application can reference a LOB value via a large object locator (LOB locator). [17]

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

For example, when selecting a LOB value, an application program could select the entire LOB value and place it into an equally large host variable (which is acceptable if the application program is going to process the entire LOB value at once), or it could instead select the LOB value into a LOB locator. Then, using the LOB locator, the application program can issue subsequent database operations on the LOB value (such as using it as a parameter to the scalar functions SUBSTR, CONCAT, COALESCE, LENGTH, doing an assignment, searching the LOB value with LIKE or POSSTR, or using it as a parameter to a user-defined function) by supplying the LOB locator value as input. The resulting output of the LOB 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 LOB locator host variable itself can never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value — the server does not track null values with valid LOB locators.

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 LOB locator, there is no

17. There is no ability within a Java application to distinguish between a CLOB or BLOB that is represented by a LOB locator and one that is not.

operation that one can perform on the original row or table that will affect the value which is referenced by the LOB locator. The value associated with a LOB locator is valid until the transaction ends, or until the LOB locator is explicitly freed, whichever comes first.

A LOB locator is 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, 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.

For the restrictions that apply to the use of LOB strings, see “Limitations on Use of Strings”.

Limitations on Use of Strings

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

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

Table 2. Contexts for limitations on use of varying-length strings

Context of usage	VARCHAR (>255) or VARGRAPHIC (>127)	LOB (CLOB, DBCLOB, or BLOB)
A GROUP BY clause	Not allowed	Not allowed
An ORDER BY clause	Not allowed	Not allowed
A CREATE INDEX statement	Not allowed	Not allowed
A SELECT DISTINCT statement	Not allowed	Not allowed
A subselect of a UNION without the ALL keyword	Not allowed	Not allowed
Predicates	Cannot be used in any predicate except EXISTS and LIKE	Cannot be used in any predicate except EXISTS, LIKE, and NULL
A *result-expression* in a CASE expression	Not allowed	Not allowed
The definition of primary, unique, and foreign keys	Not allowed	Not allowed
Check constraints	See Predicates	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 VARCHAR strings longer than 255 or VARGRAPHIC strings longer than 127, CLOB or DBCLOB strings, or both as input. See the description of the individual functions in Chapter 3, “Built-in Functions” on page 129 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".

Date

A *date* is a three-part value (year, month, and day) designating a point in time under the Gregorian calendar, which is assumed to have been in effect from the year 1 A.D. [18] The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to *x*, where *x* is 28, 29, 30, or 31, depending on the month and year.

The length of a DATE column as described in the SQLDA is 10 bytes, which is the appropriate length for a character-string representation of the value.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB.

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 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.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB.

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 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.

A character-string representation must have an actual length that is not greater than 255 bytes and must not be a CLOB.

Datetime Host Variables

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

String Representations of Datetime Values

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 strings. These representations directly concern the user of SQL since for many host languages there are no

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

constants or variables whose data types are DATE, TIME, or TIMESTAMP. Thus, to be retrieved, a datetime value must be assigned to a character-string variable. 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 mechanism to specify the default date format and default time format is product-specific.

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp before the operation is performed. The *default date format* and *default time format* specifies the date and time format that will be used to interpret the string. If the CCSID of the string is not the same as the default CCSID for SBCS data, the string 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 8 characters. Trailing blanks can be included. Leading zeros can be omitted from the month and day portions. Valid string formats for dates are listed in Table 3. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use.

Table 3. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization	ISO	yyyy-mm-dd	1987-10-12
IBM USA standard	USA	mm/dd/yyyy	10/12/1987
IBM European standard	EUR	dd.mm.yyyy	12.10.1987
Japanese industrial standard Christian era	JIS	yyyy-mm-dd	1987-10-12

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 seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13:30 is equivalent to 13:30:00. Although all products accept times of 24:00:00, the handling of such times during arithmetic operations is product-specific.

Valid string formats for times are listed in Table 4. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use.

Table 4. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization	ISO	hh.mm.ss [19]	13.30.05
IBM USA standard	USA	hh:mm AM or PM	1:30 PM
IBM European standard	EUR	hh.mm.ss	13.30.05
Japanese industrial standard Christian era	JIS	hh:mm:ss	13:30:05

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

USA Format	24-Hour Clock
12:01 AM through 12:59 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.

Timestamp strings: A string representation of a timestamp is a character string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss.nnnnnn*. Trailing blanks can be included. Leading zeros can be omitted from the month, day, and hour part of the timestamp. Trailing zeros can be truncated or omitted entirely from microseconds. If any trailing digit of the microseconds portion is omitted, an implicit specification of 0 is assumed. Thus, *1990-3-2-8.30.00.10* is
G equivalent to *1990-03-02-08.30.00.100000*. Although all products accept timestamps
G whose time part is 24.00.00.000000, the handling of such timestamps during
G arithmetic operations is product-specific.

User-Defined Types

Distinct Types

A *distinct type* is a user-defined data type that shares its internal representation with a built-in 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 using "CREATE DISTINCT TYPE" on page 304.

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 types allows functions to be created specifically for AUDIO and assures that these functions cannot be applied to other types (such as pictures or text).

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:

19. This is an earlier version of the International Standards Organization format. The JIS format is equivalent to the current International Standards Organization format.

- In a CREATE DISTINCT TYPE or the object of a DROP, COMMENT, GRANT, or REVOKE statement, the normal process of qualification by authorization ID is used to determine the schema name.
- In any other context, the SQL path is used to determine the schema name. The schemas in the SQL path are searched, in sequence, and the first schema in the SQL path is selected such that the distinct type exists in the schema and the user has authorization to use the type. For a description of the SQL path, see "SQL Path" on page 38.

A distinct type does not automatically acquire the functions and operators of its source type because they might not be meaningful. (For example, it might make sense for a "length" function of the AUDIO type to return the length in seconds rather than in bytes.) Instead, distinct types support strong typing. *Strong typing* ensures that only the functions and operators that are explicitly defined on a distinct type can be applied to that distinct type. However, a function or operator of the source type can be applied to the distinct type by creating an appropriate user-defined function. The user-defined function must be sourced on the existing function that has the source type as a parameter. For example, the following series of SQL statements shows how to create a distinct type named MONEY based on data type DECIMAL(9,2), how to define the + operator for the distinct type, and how the operator might be applied to the distinct type:

```
CREATE 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.

The comparison operators are automatically generated for distinct types, except those that are sourced on a CLOB, DBCLOB, or BLOB. In addition, functions are automatically generated for every distinct type that support casting from the source type to the distinct type and from the distinct type to the source type. For example, for the AUDIO type created above, these are 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, a precedence order exists where 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 ordering. For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE PRECISION; but CLOB is NOT promotable to VARCHAR.

The database manager considers the promotion of data types when:

- performing function resolution (see "Function Resolution" on page 92)
- casting distinct types (see "Casting Between Data Types" on page 54)
- assigning built-in data types to distinct types(see "Distinct type Assignments" on page 63).

For each data type, Table 5 shows the precedence list (in order) that the database manager uses to determine the data types to which a given 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 5. Data Type Precedence Table

Data Type	Data Type Precedence List (in best-to-worst order)
SMALLINT	SMALLINT, INTEGER, decimal, real, double
INTEGER	INTEGER, 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
udt	udt (same name)

Table 5. 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 defined in "Promotion of Data Types" on page 53) is one example 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. The cast functions or CAST specification (see "CAST Specification" on page 109) can be used to explicitly change a data type. The database manager might implicitly cast data types during assignments that involve a distinct type (see "Distinct type Assignments" on page 63). In addition, when a sourced user-defined function is created, the data types of the parameters of the source function must be castable to the data types of the function that is being created (see "CREATE FUNCTION (Sourced)" on page 325).

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 61).

If truncation occurs when casting to a binary string, an error is returned.

For casts that involve a distinct type as either the data type to be cast to or from, Table 6 shows the supported casts. For casts between built-in data types, Table 7 on page 56 shows the supported casts.

Table 6. Supported casts when a distinct type is involved

Data type ...	Is castable to data type ...
Distinct type *DT*	Source data type of distinct type *DT*
Source data type of distinct type *DT*	Distinct type *DT*
Distinct type *DT*	Distinct type *DT*

Table 6. Supported casts when a distinct type is involved (continued)

Data type ...	Is castable to data type ...
Data type *A*	Distinct type *DT* where *A* is promotable to the source data type of distinct type *DT* (see "Promotion of Data Types" on page 53)
INTEGER	Distinct type *DT* if *DT*'s source data type is SMALLINT
DOUBLE	Distinct type *DT* if *DT*'s source data type is REAL
VARCHAR	Distinct type *DT* if *DT*'s source data type is CHAR or GRAPHIC
VARGRAPHIC	Distinct type *DT* if *DT*'s source data type is GRAPHIC or CHAR

Character and graphic strings are only compatible for Unicode data.

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 92, and "CAST Specification" on page 109. 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 7. Supported Casts between Built-in Data Types

Source Data Type ↓ / Target Data Type →	SMALLINT	INTEGER	DECIMAL	NUMERIC	REAL	DOUBLE	CHAR	VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	DBCLOB	BLOB	DATE	TIME	TIMESTAMP
SMALLINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
NUMERIC	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	-	-	Y	Y	Y	-	Y[1]	-	Y	Y	Y	Y
VARCHAR	Y	Y	Y	Y	-	-	Y	Y	Y	-	Y[1]	-	Y	Y	Y	Y
CLOB	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-
GRAPHIC	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-
VARGRAPHIC	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-
DBCLOB	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-
DATE	-	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	-	-
TIME	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-
TIMESTAMP	-	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	Y	Y

Notes

[1] The cast is only allowed if the encoding scheme of the data type is Unicode.

The following table indicates where to find the rules that apply for each cast:

Table 8. Rules for Casting to a Data Type

Target Data Type	Rules
SMALLINT	See “SMALLINT” on page 209.
INTEGER	See “INTEGER or INT” on page 179.
DECIMAL	See “DECIMAL or DEC” on page 167.
NUMERIC	See “DECIMAL or DEC” on page 167.
REAL	See “REAL” on page 202.
DOUBLE	See “DOUBLE_PRECISION or DOUBLE” on page 171.
CHAR	See “CHAR” on page 150.
VARCHAR	See “VARCHAR” on page 225.
CLOB	See “CLOB” on page 155.

Table 8. Rules for Casting to a Data Type (continued)

Target Data Type	Rules
GRAPHIC	See the rules for string assignment to a host variable in "Assignments and Comparisons" on page 58.
VARGRAPHIC	See the rules for string assignment to a host variable in "Assignments and Comparisons" on page 58.
DBCLOB	See "DBCLOB" on page 166.
BLOB	See "BLOB" on page 148.
DATE	See "DATE" on page 159.
TIME	See "TIME" on page 215.
TIMESTAMP	If the source data type is a character string, see "TIMESTAMP" on page 216, where one operand is specified.

Assignments and Comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of statements such as CALL, INSERT, UPDATE, FETCH, SELECT INTO, and VALUES INTO. 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 and functions such as COALESCE and CONCAT. The compatibility matrix is as follows:

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

Note:

1 Bit data and graphic strings are not compatible. For DB2 UDB for UWO, character strings and graphic strings are only compatible in a Unicode database.

2 All character strings, even those defined with the FOR BIT DATA attribute, are not compatible with binary strings.

3 The compatibility of datetime values and character strings is limited to assignment and comparison:

- Datetime values can be assigned to character-string columns and to character-string variables as explained in "Datetime Assignments" on page 62.
- A valid string representation of a date can be assigned to a date column or compared with a date.
- A valid string representation of a time can be assigned to a time column or compared with a time.
- A valid string representation of a timestamp can be assigned to a timestamp column or compared with a timestamp.

4 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 63.

A basic rule for assignment operations is that a null value cannot be assigned to:

- a column that cannot contain null values
- a host variable that does not have an associated indicator variable
- a Java host variable that is a primitive type.

See "References to Host Variables" on page 85 for a discussion of indicator variables. For any comparison where nulls are involved, see the description of the operation for the specific handling of null values.

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. [20]

Decimal or Integer to Floating-Point

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

Floating-Point or Decimal to Integer

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

Decimal to Decimal

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

Integer to Decimal

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

Floating-Point to Decimal

When a floating-point number is assigned to a decimal column or variable, the number is converted first to a temporary decimal number and then, if necessary, to the precision and scale of the target. The conversion from floating point to decimal involves rounding and the selection of a suitable precision and scale for the
G decimal number. The precision is product-specific and dependent on whether the
G floating-point number is a single- or double-precision number. 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 Integers

Assignment to COBOL integer variables uses the full size of the integer. Thus, the value placed in the COBOL data item field may be out of the range of values.

Examples:

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

20. If truncation happens on assignment to a host variable with an indicator variable, the indicator variable may be set to -2. See "References to Host Variables" on page 85 for more information.

```
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 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 or parameter, an error is returned.

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, a warning is returned (SQLSTATE 01004) and the value 'W' is assigned to the SQLWARN1 field of the SQLCA. For a description of the SQLCA, see Appendix C, "SQL Communication Area (SQLCA)" on page 525.

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.

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 62. 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 63.

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 the parameter. Trailing blanks are included in the length of the string. When the length of the string is greater than the length attribute of the column or the parameter, one of the following occurs:

- the string is assigned with trailing blanks truncated to fit the length attribute of the targer column or parameter
- the string is not assigned and an error is returned because truncation to fit the length attribute of the column or parameter would remove non-blank characters.

When a string is assigned to a fixed-length column or parameter and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of blanks. The pad character is always a blank, even for columns defined with the FOR BIT DATA attribute.

Retrieval Assignment: The length of a string assigned to a host variable can be greater than the length attribute of the host variable. When a string is assigned to a host variable and the length of the string is greater than the length attribute of the host variable, the string is truncated on the right by the necessary number characters. When this occurs, a warning is returned and the value 'W' is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided and the source of the value is not a LOB, the indicator variable is set to the original length of the string. The truncation result of an improperly formed mixed string is unpredictable.

When a character string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of blanks. The pad character is always a blank, even for strings defined with the FOR BIT DATA attribute.

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

Assignments Involving Mixed Strings: Assignment of a character string to a host variable may result in truncation of the mixed data string. Truncation will remove complete characters from the right side of the mixed data string. Removal of a character that is longer than a single byte may cause the length of the result string to be less than the length attribute of the host variable. If padding is then required, the single-byte blank character is used.

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*, the string is padded on the right with *x-n-1* blanks, where *x* is the length of the variable. The padded string is then assigned to the variable, and a NUL is placed in the next character position. [21]

G In DB2 UDB for z/OS and OS/390, the above is true only for fixed-length string
G columns. The value of a varying-length string column is assigned to the first *n*
G character positions of the variable, and a NUL is placed in the next character
G 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 identifies bit data
- The string is neither null nor empty.
- The CCSID Conversion Selection Table ("Coded Character Sets and CCSIDs" on page 26) indicates that conversion is necessary.

---

21. In DB2 UDB for iSeries and DB2 UDB for UWO, a program preparation option must be used for the padding and NUL placement to occur as described. For DB2 UDB for iSeries use the program preparation option *CNULRQD. For DB2 UDB for UWO, use the program preparation option LANGLEVEL SQL92E.

An error is returned if:

- The CCSID Conversion Selection Table 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 DBCS character cannot be converted to a host variable with an SBCS CCSID.

A warning occurs if:

- A character of the string is converted to 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.

G In DB2 UDB for UWO, if a character of the string cannot be converted, an error is
G returned regardless of whether an indicator variable is provided.

G In a DB2 UDB for UWO Application Server in DRDA, input host variables are
G converted to the code page of the application server, even if being assigned,
G compared, or combined with a FOR BIT DATA column. If the SQLDA has been
G modified to identify the input host variable as FOR BIT DATA, conversion is not
G performed.

Datetime Assignments

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

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

- If the target is a character column, truncation is not allowed. The length attribute of the column must be at least 10 for a date, 8 for a time, and 26 for a timestamp.
- When the target is a host variable, the following rules apply:

  DATE

  The length of the variable must not be less than 10.

  TIME

  If the USA format is used, the length of the variable must not be less than 8. This format does not include seconds.

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

TIMESTAMP

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

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:

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

When the statement is executed, the following cast functions are also generated:

```
AGE (SMALLINT) RETURNS AGE
AGE (INTEGER) RETURNS AGE
SMALLINT (AGE) RETURNS SMALLINT
```

Next, assume that column STU_AGE was defined in table STUDENTS with distinct type AGE. Now, consider this valid assignment of a student's age to host variable HV_AGE, which has an INTEGER data type:

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

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

Assignments Other Than to Host Variables

A distinct type can be 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 54 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
- 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

- 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 DECIMAL(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 9 shows whether the assignments are valid. DB2 uses assignment rules in this INSERT statement to determine if X can be assigned to Y.

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

Table 9. Asessment of various assignments for the example INSERT statement

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

Assignments Involving LOB Locators

When a LOB locator is used, it can only refer to LOB data. If a LOB locator is used for the first fetch of a cursor, LOB locators must be used for all subsequent fetches.

Numeric Comparisons

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

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

When decimal numbers with different scales are compared, the comparison is made as if one of the numbers has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.

If one number is floating point and the other is integer, decimal, or single-precision floating point, the comparison is made with a temporary copy of the other number that has been converted to double-precision floating point. However, if a

single-precision floating-point number is compared to a floating-point constant, the comparison is made with a single-precision form of the constant.

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

String Comparisons

Binary String Comparisons

In general, comparisons involving binary strings (BLOBs) are not supported, with the exception of LIKE, EXISTS and the NULL predicate.

Character and Graphic String Comparisons

Two character or graphic strings are compared by comparing the corresponding bytes of each character or graphic string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string. The pad character is always a blank, even for bit data.

The relationship between two unequal strings is determined by the comparison of the first pair of unequal bytes from the left end of the string. [22]

Note that the encoding scheme used for the data determines the sort sequence which impacts the resulting order. [23]

In an application that will run in multiple environments, the same sort sequence (which will depend on the CCSIDs of the environments) must be used to ensure identical results. The following table illustrates the differences between EBCDIC, ASCII, and the DB2 UDB for UWO default sort sequence (using SYSTEM collation TERRITORY Un_US) by showing a list sorted according to each one.

Table 10. Sort Sequence Differences

ASCII and Unicode	EBCDIC	DB2 UDB for UWO Default
0000	@@@@	0000
9999	co-op	9999
@@@@	coop	@@@@
COOP	piano forte	co-op
PIANO-FORTE	piano-forte	COOP
co-op	COOP	coop
coop	PIANO-FORTE	piano forte
piano forte	0000	PIANO-FORTE
piano-forte	9999	piano-forte

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 “group-by-clause” on page 244 for more information about the arbitrary selection involved in references to a grouping column.

22. In DB2 UDB for UWO, this is only true if a sort sequence with unique weights is chosen when the database is created.

23. In DB2 UDB for UWO, to get this behavior, specify the IDENTITY collation on CREATE DATABASE. Product-specific options are available on DB2 UDB for iSeries and DB2 UDB for UWO to change the sort sequence independent of the encoding scheme.

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 X'FFFF'.
- The string selected for conversion is neither null nor empty.
- The CCSID Conversion Selection Table (“Coded Character Sets and CCSIDs” on page 26) indicates that conversion is necessary.

If a Unicode string and a non-Unicde string are compared, any necessary conversion applies to the non-Unicode string. If an SBCS string and a MIXED string are compared, any necessary conversion applies to the SBCS string. 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.

Table 11. Selecting the Operand for Character Conversion

First Operand	Second Operand				
	Column Value	**Derived Value[24]**	**Constant**	**Special Register**	**Host Variable**
Column Value	second	second	second	second	second
Derived Value[24]	first	second	second	second	second
Constant	first	first	second	second	second
Special Register	first	first	second	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 is returned if a character of the string cannot be converted or the CCSID Conversion Selection Table (“Coded Character Sets and CCSIDs” on page 26) 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 a value 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.

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'
```

24. In DB2 UDB for z/OS and OS/390, derived values are converted before constants and special registers.

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:

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

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER                          INCORRECT
  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 function COALESCE
- Expression values of the IN list of an IN predicate

For the result data type of expressions involving the operators /, *, + and -, see "With Arithmetic Operators" on page 96. For the result data type of expressions involving the CONCAT operator, see "With the Concatenation Operator" on page 98.

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 is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
INTEGER	SMALLINT	INTEGER
INTEGER	INTEGER	INTEGER
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)	DECIMAL(y,z) or NUMERIC(y,z)	DECIMAL(p,s) where p = min(31,max(x,z)+max(w-x,y-z)) and s = max(x,z)

If one operand is...	And the other operand is...	The data type of the result is...
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)	NUMERIC(y,z)	NUMERIC(p,s) where p = min(31,max(x,z)+max(w-x,y-z)) and s = max(x,z)
REAL	REAL	REAL
REAL	DECIMAL, NUMERIC, INTEGER, or SMALLINT	DOUBLE
DOUBLE	any numeric	DOUBLE

Character String Operands

Character strings are compatible with other character strings.

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

The CCSID of the result character string will be derived based on the “Conversion Rules for Operations that Combine Strings” on page 71.

Graphic String Operands

Graphic strings are compatible with other graphic strings.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	GRAPHIC(y) OR VARGRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where z = max (x,y)

The CCSID of the result graphic string will be derived based on the “Conversion Rules for Operations that Combine Strings” on page 71.

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 is...	And the other operand is...	The data type of the result 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 character string expression that contains a valid string representation of a date. A string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is DATE.

A TIME type is compatible with another TIME type, or any character string expression that contains a valid string representation of a time. A string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIME.

A TIMESTAMP type is compatible with another TIMESTAMP type, or any character string expression that contains a valid string representation of a timestamp. A string representation must not be a CLOB and must have an actual length that is not greater than 255 bytes. The data type of the result is TIMESTAMP.

If one operand is...	And the other operand is...	The data type of the result is...
DATE	DATE, CHAR(y) or VARCHAR(y)	DATE
TIME	TIME, CHAR(y) or VARCHAR(y)	TIME
TIMESTAMP	TIMESTAMP, CHAR(y) or VARCHAR(y)	TIMESTAMP

Distinct Type Operands

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

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

Conversion Rules for Operations that Combine Strings

The operations that combine strings are concatenation, UNION, and UNION ALL. These rules also apply to the COALESCE 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 X'FFFF', the result is X'FFFF'. [25]
- If one CCSID denotes Unicode data and the other denotes non-Unicode data, the result is the CCSID for Unicode data.
- If one CCSID denotes SBCS data and the other denotes mixed data, the result is the CCSID for mixed data.
- Otherwise, the result CCSID is determined by the following table:

Table 12. 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

G G In DB2 UDB for z/OS and OS/390, derived values are converted before constants and special registers.

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 would be determined as follows:

– 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.

25. Both operands must not be a CLOB or DBCLOB.

– The result CCSID of “intermediate result” CONCAT COLC is determined to be 500, 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 COALESCE and CONCAT scalar functions or the result expression of the CASE expression 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 X'FFFF'.
- The string is neither null nor empty.
- The CCSID Conversion Selection Table (“Coded Character Sets and CCSIDs” on page 26) indicates that conversion is necessary.

An error is returned 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 pair of CCSIDs. A warning occurs if a character of a string is converted to the substitution character.

Constants

A *constant* (also 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 10 digits that does not include a decimal point. The data type of an integer constant is large integer, and its value must be within the range of a large integer.

Examples

```
62     -15     +100     32767     720176
```

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

Floating-Point Constants

A *floating-point constant* specifies a 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 2.

Examples

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

Decimal Constants

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

Examples

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

Character-String Constants

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

- It is a sequence of characters enclosed between apostrophes. The number of bytes between the apostrophes cannot be greater than 255. See Table 39 on page 510 for more information. Two consecutive apostrophes are used to represent one apostrophe within the character string, but these count as one byte when calculating lengths of character constants. Two consecutive apostrophes that are not contained within a string represent an empty string.

- An X followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 254. See Table 39 on page 510 for more information. 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.

At installations that have 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. In DB2 UDB for UWO, in a DBCS environment, all character string constants are classified as mixed data. The CCSID assigned to the constant is the appropriate default CCSID of the application server at bind time. A mixed data constant can be continued from one line to the next only if the break occurs between single-byte characters.

Character-string constants are used to represent constant datetime values in assignments and comparisons. For more information see“String Representations of Datetime Values” on page 49.

Examples

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

Graphic-String Constants

A *graphic-string constant* specifies a varying-length graphic string. The length of the specified string cannot be greater than 124. See Table 39 on page 510 for more information.

In EBCDIC environments, the forms of graphic-string constants are :

Graphic String Constant	Empty String	Example
G'$S_O$dbcs-stringS_I'	G'$S_O$$S_I$'	G'S_O元気S_I'
	G''	
	g'$S_O$$S_I$'	
	g''	
N'$S_O$dbcs-stringS_I'	N'$S_O$$S_I$'	
	N''	
	n'$S_O$$S_I$'	
	n''	

In ASCII environments, the form of the constant is:

G'*dbcs-string*' or N'*dbcs-string*'

The CCSID assigned to the constant is the appropriate default CCSID of the application server at bind time.

In SQL statements and in host language statements in a source program, graphic strings cannot be continued from one line to another.

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, and FLOAT scalar functions and the CAST specification)
- to determine the decimal point character to use in the result when casting a number to a string (for example, in the CHAR scalar function and the CAST specification)

G Each product provides a product-specific means of explicitly specifying a *default decimal point*.

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.

For portability across the platforms, when defining a variable to receive the contents of a special register that contains character data it is recommended that the variable be defined with the maximum length supported by any of the platforms for that special register. See Appendix A, "SQL Limits" on page 509 for more information on the maximum lengths of the special registers.

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

Example

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

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

CURRENT PATH

The CURRENT PATH special register specifies the SQL path used to resolve unqualified distinct type names, function names, and procedure names in dynamically prepared SQL statements. It is used to resolve unqualified procedure names that are specified as host variables in SQL CALL statements (CALL *host-variable*). The data type is VARCHAR with a length attribute that is the maximum length of a path. See Appendix A, "SQL Limits" on page 509 for more information.

The CURRENT PATH special register contains the value of the SQL path which is a list of one or more schema names, where each schema name is enclosed in delimiters and separated from the following schema name by a comma, (any delimiters within the string are repeated as they are in any delimited identifier). The delimiters and commas are included in the length of the special register.

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 "Unqualified Distinct Type, Function, Procedure, and Specific Names" on page 38.

The initial value of the special register in a user-defined function or procedure is inherited from the invoking application. In other contexts the initial value of the special register is the system path followed by the USER special register value. For more information on the system path, see "The System Path" in "SET PATH" on page 461.

The value of the special register can be changed by executing the SET PATH statement. For details about this statement, see "SET PATH" on page 461.

Example

Set the special register so that schema SMITH is searched before the system schemas:

```
SET PATH = SMITH, SYSTEM PATH;
```

CURRENT SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that
G identifies the current server. In DB2 UDB for z/OS and OS/390, CURRENT
G SERVER specifies a CHAR(16) value. See Appendix A, "SQL Limits" on page 509.

The CURRENT SERVER can be changed by the CONNECT (Type 1), CONNECT (Type 2), or SET CONNECTION statements, but only under certain conditions. See "CONNECT (Type 1)" on page 296, "CONNECT (Type 2)" on page 300, and "SET CONNECTION" on page 459 for more information.

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

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

Example

Using the CL_SCHED sample table, select all the classes (CLASS_CODE) that start (STARTING) later today. Today's classes have a value of 3 in the DAY column.

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

CURRENT TIMESTAMP

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

Example

Insert a row into the IN_TRAY 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

The CURRENT TIMEZONE special register specifies the difference between UTC [26] 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. The data type of
G the register is VARCHAR(18). In DB2 UDB for z/OS and OS/390 the data type is
G CHAR(8). See Appendix A, “SQL Limits” on page 509.

Example

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

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

26. 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 233.) For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a *GROUP BY or ORDER BY clause,* a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
  - In an *expression,* a *search condition,* or a *scalar function,* a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Provide a column name for an expression, temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause, or in the AS clause in the *select-clause*.

Qualified Column Names

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

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

- In the COMMENT statement specifying ON COLUMN, the column name must be qualified.
- Where the column name specifies values of the column, a column name may be qualified.
- In the *assignment-clause* of an UPDATE statement, it may be qualified.
- 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 81 and "Column Name Qualifiers in Correlated References" on page 83 for details.

Correlation Names

A *correlation name* can be defined in the FROM clause of a query and after the target *table-name* or *view-name* in 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 or view only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements, or in different clauses of the same statement.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table or view. In the example 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 name or view name, any qualified reference to a column of that instance of the table or view must use the correlation name, rather than the table name or view 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 *nonexposed*. A correlation name is always an exposed name. A table name or view name is said to be *exposed* in that FROM clause if a correlation name is not specified. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
FROM EMPLOYEE E, DEPARTMENT
```

A table name or view 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
    WHERE EMPLOYEE.PROJECT = 'ABC'
```

INCORRECT

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

5. Given the FROM clause:

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

a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). 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
- The second SQL identifier of any qualified table name or view name that is exposed in the FROM clause.

For example, the following FROM clauses are incorrect:

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

INCORRECT

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 target table or view in a DELETE or UPDATE statement or *table-reference* in a FROM clause. The tables, views and *table-reference*s [27] that might contain the column are

27. In the case of a *joined-table*, each *table-reference* within the *joined-table* is an object table.

called the *object tables* of the context. Two or more object tables might contain columns with the same name. One reason for qualifying a column name is to designate the object from which the column comes. For information on avoiding ambiguity between SQL parameters and variables and column names, see “References to SQL Parameters and SQL Variables” on page 478.

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.
- 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.

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 EMPLOYEE:

```
SELECT * FROM EMPLOYEE X,EMPLOYEE Y
```

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. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name may be used. However, the qualifier used and the table used must be the same after fully qualifying the table name or view name and the table designator.

1. If the authorization ID of the statement is CORPDATA:

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

   is a valid statement.

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

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

is invalid, because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE.

Column Name Qualifiers in Correlated References

A *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 233 for more information on subselects. A *subquery* is a form of a fullselect that is enclosed within parenthesis. For example, a subquery can be used in a search condition. A fullselect used in the FROM clause of a query is called a *nested table expression*.

A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. 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 of a subquery can reference not only columns of the tables identified by the FROM clause of its own element of the hierarchy, but also columns of tables identified at any level along the path from its own element to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a *correlated reference*.

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

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

- Q.C is used in a search condition of a subquery
- Q does not designate an exposed table used in the FROM clause of that subquery
- Q does designate an exposed 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 in Correlated References

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. When an SQL statement that has an unqualified correlated reference is prepared or executed, a warning is returned.

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 *variable*s 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 85.

transition variable
: Transition variables are defined in a trigger and refer to either the old or new values of columns of the subject table of a trigger. For more information about how to refer to transition variables see "CREATE TRIGGER" on page 368.

SQL variable
: SQL variables are defined by an SQL compound statement in an SQL procedure. For more information about SQL variables, see "References to SQL Parameters and SQL Variables" on page 478.

SQL parameter
: SQL parameters are defined in an CREATE FUNCTION (SQL Scalar) or CREATE PROCEDURE (SQL) statement. For more information about SQL parameters, see "References to SQL Parameters and SQL Variables" on page 478.

parameter marker
: Parameter markers are specified in an SQL statement that is dynamically prepared instead of host variables. For more information about parameter markers, see "Notes" on page 434 in the PREPARE statement.

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 or a C[28], Java, or REXX variable that is referenced in an SQL statement. Host variables are defined by statements of the host language. Host variables cannot be referenced in dynamic SQL statements; instead, parameter markers must be used. For more information on parameter markers, see "Host Variables in Dynamic SQL" on page 87.

A *host-variable* in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

All host variables used in an SQL statement must be declared in an SQL declare section in all host languages other than Java and REXX. Variables do not have to be declared in REXX. In Java, variables must be declared, but an SQL declare section is not necessary or allowed. 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.

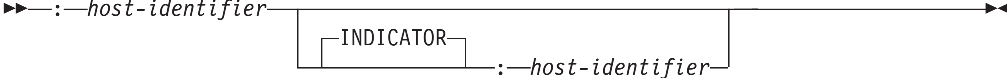
For further information about using host variables, see:

28. In this book, whenever the C language is referenced, the information also applies to C++.

- Appendix H, "Coding SQL Statements in C Applications" on page 593
- Appendix I, "Coding SQL Statements in COBOL Applications" on page 609
- Appendix J, "Coding SQL Statements in Java Applications" on page 627
- Appendix K, "Coding SQL Statements in REXX Applications" on page 643

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, a SELECT INTO, or a VALUES INTO statement identifies a host variable to which a column value is assigned. A host variable in a CALL statement can be an output argument that is assigned a value after execution of the procedure, an input argument that provides an input value for the procedure, or both an input and output argument. In all other contexts a *host-variable* specifies a value to be passed to the database manager 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 be a small integer.

The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value.
- Indicate that a numeric conversion error (such as a divide by 0 or overflow) has occurred. [29]
- Indicate that a character could not be converted.
- Record the original length of a truncated string, if the string is not a LOB.
- 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 if 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
- -2 if the null value was returned due to a numeric conversion error (such as divide by 0 or overflow) or a character conversion error. [30]

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

---

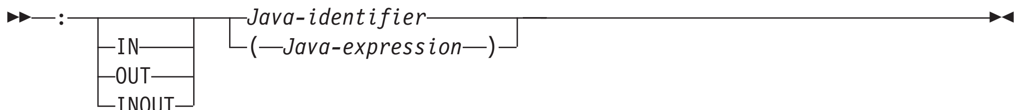
29. In DB2 UDB for UWO, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

30. It should be noted that although a -2 null value can be returned for conversion errors, the result column itself is not considered nullable unless an argument of the expression, scalar function, the column is nullable.

the original length of the string). If an assignment requires truncation of the seconds part of a time, V2 is set to the number of seconds.

If the second *host-identifier* is omitted, the host variable does not have an indicator variable. The value specified by the *host-variable* :V1 is always the value of V1, and null values cannot be assigned to the variable. Thus, this form should not be used unless the corresponding result column cannot contain null values. If this form is used and the column contains nulls, the database manager will return an error at run-time.

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 “Using Host Variables and Expressions in Java” on page 631.

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 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. In this case the host variable value 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 statement that uses host variables and a dynamic statement that uses parameter markers:

References to Host Variables

```
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 “PREPARE” on page 433.

References to LOB Host Variables

Regular LOB variables can be defined in all host languages other than REXX. LOB locator variables can be defined in the following host languages:

- C
- COBOL

Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable or a locator variable. Since these variables are not native data types in host programming languages, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

When it is possible to define a host variable that is large enough to hold an entire LOB value and the performance benefit of delaying the transfer of data from the server is not required, a LOB locator is not needed. However, it is often not acceptable to store an entire LOB value in temporary storage due to host language restrictions, storage restrictions, or performance requirements. When storing an entire LOB value at one time is not acceptable, a LOB value can be referenced using a LOB locator and portions of the LOB value can be accessed.

Like all other host variables, a LOB locator variable can have an associated indicator variable. Indicator variables for LOB locator variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the 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
- COBOL

See “Manipulating Large Objects (LOBs) with Locators” on page 47 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.

Host Structures

A *host structure* is a C structure or COBOL group, that is referred to in an SQL statement. In Java and REXX, there is no equivalent to a *host structure*. Host structures are defined by statements of the host language. As 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 referred to in any context where a list of host variables can be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator array is the indicator variable for the *n*th variable of the host structure.

In C, for example, if V1, V2, and V3 are declared as the variables within the structure S1, the statement:

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

is equivalent to:

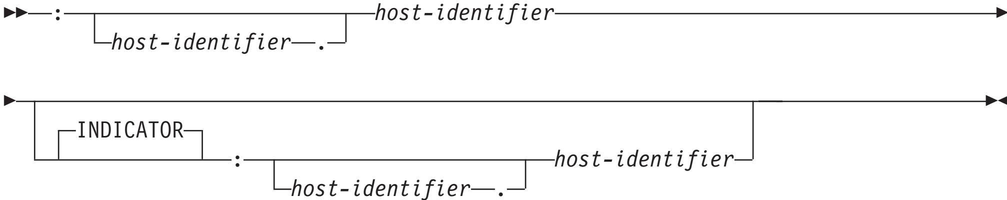
```
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

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 variable is not specified, no variable of the host structure has an indicator variable.

In addition to structure references, individual host variables in a host structure or indicator variables in an indicator array can be referred to by qualified names. The qualified form is a host identifier followed by a period and another host identifier. The first host identifier must name a structure and the second host identifier must name a host variable within that structure.

The general form of a host variable or host structure reference is:

Host Structures in C and COBOL



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
```

Functions

A *function* is an operation denoted by a function name followed by zero or more operands that are enclosed in parentheses. It represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, a function can be passed two input arguments that have date and time data types and return a value with a timestamp data type as the result.

Types of Functions

There are several ways to classify functions. One way to classify functions is as built-in, user-defined, or generated user-defined functions for distinct types.

- *Built-in functions* are 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 129.

  G The *built-in functions* are in a product-specific schema.
- *User-defined functions* are functions that are created using the CREATE FUNCTION statement and registered to the database manager in the catalog. For more information, see “CREATE FUNCTION” on page 310. 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 an *SQL*, *external*, or *sourced* function. 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 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.
- *Generated user-defined functions for distinct types* are functions that the database manager automatically generates 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 user-defined functions for distinct types reside in the same schema as the distinct type for which they were created. For more information about the functions that are generated for a distinct type, see “CREATE DISTINCT TYPE” on page 304.

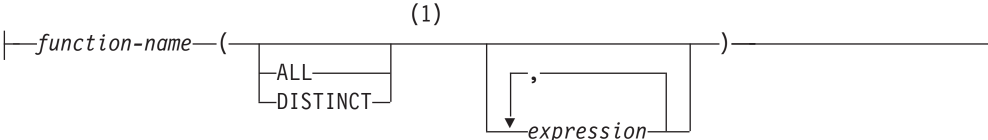
Another way to classify functions is as column or scalar functions, depending on the input data values and result values.

- 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.
- 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. Generated user-defined functions for distinct types are also scalar functions.

Function Invocation

Each reference to a function conforms to the following syntax:[31]

function-invocation:



Notes:

1 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.

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 selects candidate functions 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 arguments in the function invocation.
  - The authorization ID of the statement must have the EXECUTE privilege to the function instance.
  - The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

  If no function in the schema meets these criteria, an error is returned.
- *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 SQL path (see "SQL Path" on page 38), the database manager selects candidate functions based on the following criteria:

31. A few functions allow keywords instead of a comma separated list of expressions. For example, the CHAR function allows a list of keywords to indicate the desired date format.

– The name of the function instance matches the name in the function invocation.
– The number of input parameters in the function instance matches the number of function arguments in the function invocation.
– The authorization ID of the statement must have the EXECUTE privilege to the function instance.
– The data type of each input argument of the function invocation matches or is *promotable* to the data type of the corresponding parameter of the function instance.

If no function in the schema meets these criteria, an error is returned.

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 the system portion of the SQL path. For more information on the system portion of the SQL path, see "SQL Path" on page 38. When an unqualified function name is specified, the SQL path must be set to a list of schemas in the desired search order 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 the data type of the result of the function or the type of function (column, scalar, or table) under consideration does not enter into this determination.

If the data types of all the parameters for a given function are the same as those of the arguments in the function invocation, that function is the best fit. If there is no exact match, 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 53 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 92. 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, NUMERIC(8,0))...
4. CREATE FUNCTION ROMANO.MYFUNC (INT, FLOAT) ...
```

Also assume that the SQL path at the time the application invokes the function is ″ROMANO″, ″KNAPP″, ″BESTGEN″ and that the authorization ID of the statement has the EXECUTE privilege to functions 1, 2 and 4. 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:

- Function 3 is eliminated. It is not a candidate for this function invocation because the authorization ID of the statement does not have the EXECUTE privilege to the function. The remaining three functions are candidates for this function invocation because they meet the criteria specified in “Function Resolution” on page 92.
- Function 4 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.

Best Fit Considerations

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 is returned. 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):

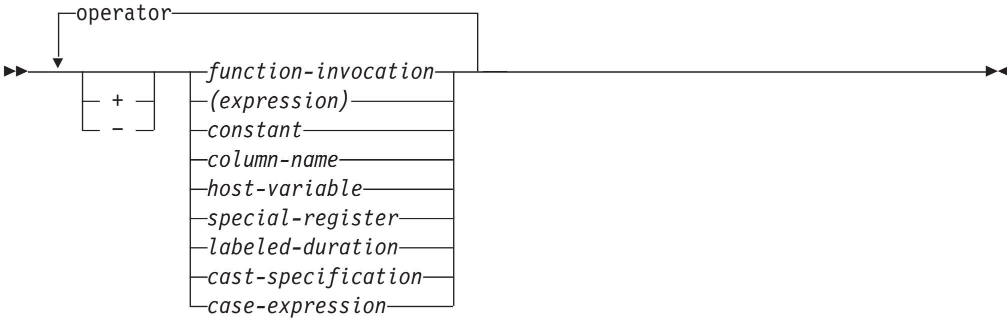
```
SELECT ... 3 +STEP(S)
```

then, because there is an exact match on argument type, the first STEP is chosen. An error is returned 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 storage assignment rules (see “Assignments and Comparisons” on page 58). This includes the case where precision, scale, length, or CCSID differs between the argument and the parameter.

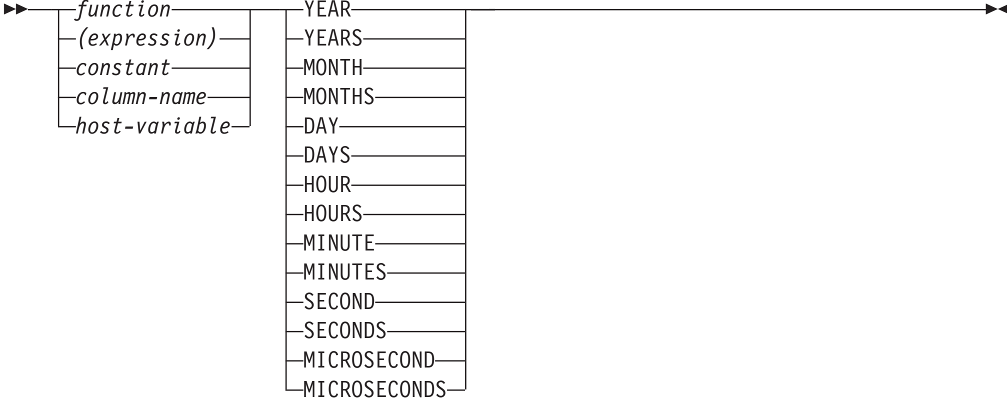
Expressions

An expression specifies a value.



operator:

labeled-duration:



Without Operators

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

Examples

```
SALARY      :SALARY      'SALARY'      MAX(SALARY)
```

With Arithmetic Operators

If arithmetic operators are used, the result of the expression is a number derived from the application of the operators to the values of the operands.

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, and division, respectively. If the value of the second operand of division is zero, then an error is returned.

Two Integer Operands

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

Integer and Decimal Operands

If one operand is an integer and the other is decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with precision *p* and scale 0. *p* is 11 for a large integer and 5 for a
G small integer. However, in the case of an integer constant, *p* is product-specific.

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 as if a temporary copy of the operand with the smaller scale is made by extending it with trailing zeros so that its fractional part has the same scale 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). [32]

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).

Multiplication: The precision of the result of multiplication is min (31,*p*+*p*′) and the
G scale is min(31,*s*+*s*′). In DB2 UDB for z/OS and OS/390, special rules apply if both
G p and p′ are greater than 15. See the product reference for further information.

32. For DB2 UDB for z/OS and OS/390, the formulas used in this book are those that apply when the DEC31 option is in effect or the precision of an operand is greater than 15.

Division: The precision of the result of division is 31. The scale is *31-p+s-s*'. The
G scale must not be negative. In DB2 UDB for z/OS and OS/390, the scale is
G different and special rules apply when p' is greater than 15. See the product
G reference for further information.

Floating-Point Operands

If either operand of an arithmetic operator is floating point, the operation is performed in floating point, the operands having been 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 that has been converted to double-precision floating point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number that has been converted to double-precision floating point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can 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 database manager (for example, the database manager may decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

Distinct type Operands

A distinct type cannot be used with arithmetic operators even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, 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)
```

Alternatively, the distinct type can be cast to a built-in type and the result used as an operand of an arithmetic operator.

With the Concatenation Operator

If the concatenation operator (CONCAT or ||) is used, the result of the expression is a string.[33]

33. Using the vertical bar (|) character might inhibit code portability between DB2 relational database products. Use the CONCAT operator in place of the || operator. On the other hand, if conformance to SQL 1999 Core standard is of primary importance, use the || operator).

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 13. 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 = MIN(x + y, maximum length of a DBCLOB)
VARGRAPHIC(x)	CHAR(y)* or VARCHAR(y)* or GRAPHIC(y) or VARGRAPHIC(y)	VARGRAPHIC(z) *** where z = MIN(x + y, maximum length of a VARGRAPHIC)
GRAPHIC(x)	CHAR(y)* or GRAPHIC(y)	GRAPHIC(z)** where z = MIN(x + y, maximum length of a VARGRAPHIC)
CLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	CLOB(z) where z = MIN(x + y, maximum length of a CLOB)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) **** where z = MIN(x + y, maximum length of a VARCHAR)
CHAR(x)	CHAR(y)	CHAR(z)** where z = MIN(x + y, maximum length of a VARCHAR)
BLOB(x)	BLOB(y)	BLOB(z) where z = MIN(x + y, 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.

** In EBCDIC environments, if either operand is mixed data, the resulting data type is VARCHAR(z).

*** In DB2 UDB for UWO, if z evaluates to greater than 2000, a LONG VARGRAPHIC is returned.

**** In DB2 UDB for UWO, if z evaluates to greater than 4000, a LONG VARCHAR is returned.

Table 14. 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 14. 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 both operands are strings, the sum of their lengths must not exceed the maximum length of the resulting data type. See Table 39 on page 510 for more information.

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 EBCDIC 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, while the second operand is a character string beginning with a "shift-out" character, these two bytes are eliminated from the result.

The length of the result is the sum of the lengths of the operands, unless redundant shift codes are eliminated, in which case the length is two less than the sum of the lengths of the operands.

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 71. 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.

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 in "Expressions" on page 96)

A *labeled duration* represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS [34] . 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.

34. Note that the singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

Date Duration

A *date duration* represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. The result of subtracting one DATE value from another, as in the expression HIREDATE - BIRTHDATE, is a date duration.

Time Duration

A *time duration* represents a number of hours, minutes, and seconds, expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss* where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The result of subtracting one TIME value from another is a time duration.

Timestamp duration

A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *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.
- 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) <= 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) <= 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).
```

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 a warning indicator in the SQLCA is set 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 a warning indicator in the SQLCA is set 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 or subtracting a 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.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. `TIME1 + X`, where "X" is a DECIMAL(6,0) number, is equivalent to the expression:

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

Timestamp Arithmetic

Timestamps can be subtracted, incremented, or decremented.

Subtracting timestamps: The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is DECIMAL(20,6). If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = TS1 - TS2.

If MICROSECOND(TS2) <= MICROSECOND(TS1)
then MICROSECOND(RESULT) = MICROSECOND(TS1) -
MICROSECOND(TS2).

If MICROSECOND(TS2) > MICROSECOND(TS1)
then MICROSECOND(RESULT) = 1000000 +
MICROSECOND(TS1) - MICROSECOND(TS2)
and SECOND(TS2) is incremented by 1.

The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

If HOUR(TS2) <= HOUR(TS1)
then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).

If HOUR(TS2) > HOUR(TS1)
then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
and DAY(TS2) is incremented by 1.

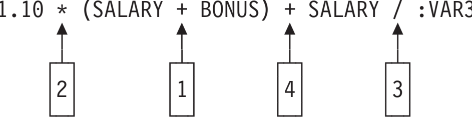
The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

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.

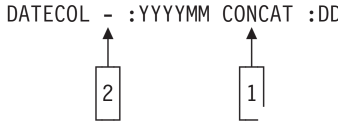
Precedence of Operations

Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication, division, and concatenation are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

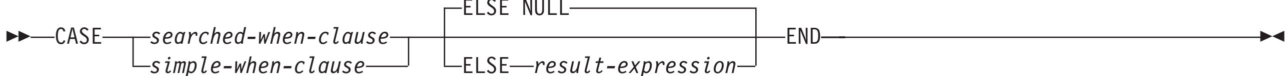
Example 1: In this example, the first operation is the addition in (SALARY + BONUS) because it is within parenthesis. The second operation is multiplication because it is at a higher precedence level than the second addition operator and it is to the left of the division operator. The third operation is division because it is at a higher precedence level than the second addition operator. Finally, the remaining addition is performed.



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.

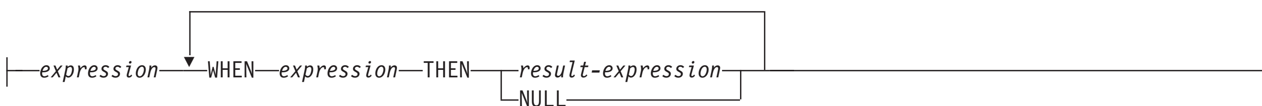


CASE Expressions



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 if 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 be a a CLOB, DBCLOB or BLOB or a character string with a maximum length greater than 254 or a graphic string with a maximum length greater than 127.
- 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 68.

search-condition
Specifies a condition that is true, false, or unknown about a row or group of table data.

The *search-condition* cannot contain a subselect.

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 15. 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, FIRSTNME, 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:

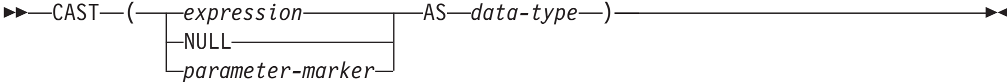
```
SELECT EMPNO, WORKDEPT, SALARY+COMM
FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN NULL
            ELSE COMM/SALARY
       END) > 0.25
```

- The following CASE expressions are equivalent:

```
SELECT LASTNAME,
 CASE
 WHEN LASTNAME = 'Haas' THEN 'President'
 ...
 ELSE 'Unknown'
 END
FROM EMPLOYEE
```

```
SELECT LASTNAME,
 CASE LASTNAME
 WHEN 'Haas' THEN 'President'
 ...
 ELSE 'Unknown'
 END
FROM EMPLOYEE
```

CAST Specification



data-type:

```
├──┬─built-in-type──────┬───────────────────────────────────────────────┤
   └─distinct-type-name─┘
```

built-in-type:

```
├──┬──┬─SMALLINT─┬───────────────────────────────────┬──────────────────┤
   │  ├─INTEGER──┤                                   │
   │  └─INT──────┘                                   │
   │                  ┌─(5,0)─────────────────────┐  │
   ├──┬─┬─DECIMAL─┬─┬─┴─(──integer──┬─,0────────┬──)┴──┤
   │  │ └─DEC─────┘ │               └─, integer─┘    │
   │  └─NUMERIC─────┘                                │
   │          ┌─(53)────────────┐                    │
   ├──┬─FLOAT─┴─(──integer──)───┴─┬──────────────────┤
   │  ├─REAL──────────────────────┤                  │
   │  │      ┌─PRECISION─┐        │                  │
   │  └─DOUBLE┴───────────┴───────┘                  │
   │                       ┌─(1)─────────────┐       │
   ├──┬──┬──┬─CHARACTER─┬──┴─(──integer──)───┴──┬─┬──┤
   │  │  │  └─CHAR──────┘                       │ │  │
   │  │  └──┬──┬─CHARACTER─┬──VARYING─┬──(──integer──)─┘ │
   │  │     │  └─CHAR──────┘          │             │  │
   │  │     └─VARCHAR─────────────────┘             │  │
   │  └─CLOB──(──integer──┬───┬──)──────────────────┘  │
   │                      ├─K─┤                        │
   │                      ├─M─┤                        │
   │                      └─G─┘                        │
   │             ┌─(1)─────────────┐                   │
   ├──┬─GRAPHIC──┴─(──integer──)───┴─┬─────────────────┤
   │  ├─VARGRAPHIC──(──integer──)────┤                 │
   │  └─DBCLOB──(──integer──┬───┬──)─┘                 │
   │                        ├─K─┤                      │
   │                        ├─M─┤                      │
   │                        └─G─┘                      │
   ├─BLOB──(──integer──┬───┬──)────────────────────────┤
   │                   ├─K─┤                           │
   │                   ├─M─┤                           │
   │                   └─G─┘                           │
   └──┬─DATE──────┬────────────────────────────────────┘
      ├─TIME──────┤
      └─TIMESTAMP─┘
```

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 EXECUTE on

the generated user-defined functions for the distinct type. If the data type of the second 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 "Casting Between Data Types" on page 54, 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.

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 storage assignment rules, see "Assignments and Comparisons" on page 58). Such a parameter marker is called a *typed parameter marker*. Typed parameter markers are 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 Distinct Type, Function, Procedure, and Specific Names" on page 38. For a description of *data-type*, see "CREATE TABLE" on page 353. (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 7 on page 56 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.

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 54.

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 is sourced on SMALLINT and is the data type for the AGE column in the PERSONNEL table. R_YEAR is 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 )
```

The first parameter is an untyped parameter marker that would have a data type of R_YEAR. An explicit CAST specification is not required in this case because the parameter marker value is assigned to the distinct type.

The second parameter marker is a typed parameter marker that is cast to distinct type T_AGE. An explicit CAST specification is required in this case because the parameter marker value is compared to the distinct type.

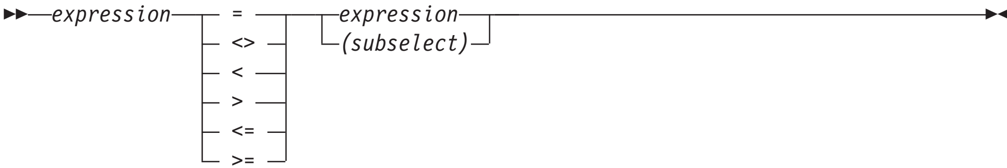
Predicates

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:

- Predicates are evaluated after the expressions that are operands of the predicate.
- All values specified in the same predicate must be compatible.
- Except for EXISTS and the first operand of LIKE, the operand of a predicate must not be a character string with a maximum length greater than 255, a graphic string with a maximum length greater than 127, or a LOB. However, a LOB can be used in a NULL predicate.
- The value of a host variable may be null (that is, the variable may have a negative indicator variable).
- The CCSID conversion of operands of predicates involving two or more operands are done according to "Conversion Rules for Comparison" on page 66.

Basic Predicate



A *basic predicate* compares two values.

A subselect in a basic predicate must specify a single result column and must not return more than one value.

If the value of either operand is null or the result of the subselect is empty, the result of the predicate is unknown. Otherwise the result is either true or false.

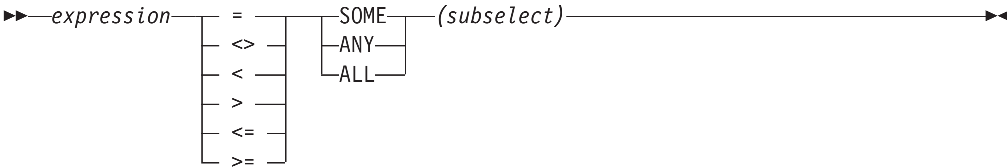
For values *x* and *y*:

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

Examples

```
EMPNO = '528671'

PRTSTAFF <> :VAR1

SALARY + BONUS + COMM < 20000

SALARY > (SELECT AVG(SALARY)
          FROM EMPLOYEE)
```

Quantified Predicate



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.

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**

```
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.

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 logically equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3)
```

that is,

```
value1 < value2 OR value1 > value3
```

G
G

If the operands of the BETWEEN predicate are strings with different CCSIDs, product-specific rules are used to determine which operands are converted.

Given a mixture of datetime values and string representations of datetime values, all operands 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──(──subselect──)──────────────────────────────────────────►◄
```

The EXISTS predicate tests for the existence of certain rows. The subselect may specify any number of columns, and

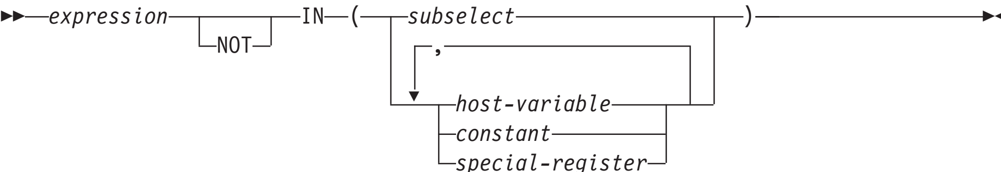
- The result is true only if the number of rows specified by the subselect is not zero
- The result is false only if the number of rows specified by the subselect is zero
- The result cannot be unknown.

Any values that may be 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.

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.

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 68.

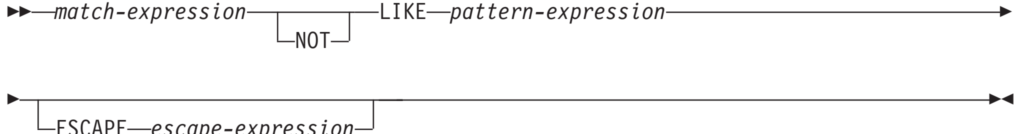
If the operands of the IN predicate are strings with different CCSIDs, the rules used to determine which operands are converted are those for operations that combine strings. For a description, see “Conversion Rules for Operations that Combine Strings” on page 71.

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 a string. The values for *match-expression*, *pattern-expression*, and *escape-expression* must either all be binary strings or none can be binary strings.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source data type.

With character strings, the terms *character*, *percent sign*, and *underscore* in the following description refer to single-byte characters. With graphic strings, the terms refer to double-byte or Unicode 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.

Simple description: A simple description of the LIKE pattern is as follows:

- The underscore sign (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

Rigorous description: Let *x* denote a value of *match-expression* and *y* denote the value of *pattern-expression*.

The string *y* is interpreted as a sequence of the minimum number of substring specifiers so each character of *y* 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 *x* or *y* is the null value. Otherwise, the result is either true or false. The result is true if *x* and *y* are both empty strings or if there exists a partitioning of *x* into substrings such that:

- A substring of *x* is a sequence of zero or more contiguous characters and each character of *x* is part of exactly one substring.
- If the *n*th substring specifier is an underscore, the *n*th substring of *x* is any single character.

- If the *n*th substring specifier is a percent sign, the *n*th substring of *x* 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 *x* is equal to that substring specifier and has the same length as that substring specifier.
- The number of substrings of *x* is the same as the number of substring specifiers.

It follows that if *y* is an empty string and *x* is not an empty string, the result is false. Similarly, it follows that if *y* is an empty string and *x* is not an empty string consisting of other than percent signs, the result is false.

The predicate *x* `NOT LIKE` *y* is equivalent to the search condition `NOT(`*x* `LIKE` *y*`)`.

If the CCSID of either the pattern value or the escape value is different than that of the column, that value is converted to adhere to the CCSID of the column before the predicate is applied.

Mixed data: If the column is mixed data, the pattern can include both SBCS and DBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one DBCS character.
- A percent (either SBCS or DBCS) refers to any number of characters of any type, either SBCS or DBCS.

In EBCDIC environments, any redundant shifts in either the column values or the pattern are ignored. [35]

If the pattern is improperly formed mixed data, the result is unpredictable.

For Unicode, the special characters in the pattern are interpreted as follows:

- An SBCS or DBCS underscore refers to one character (either SBCS or MBCS)
- A percent sign (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

When the LIKE predicate is used with Unicode data, the Unicode percent sign and underscore use the code points indicated in the following table:

Table 16.

Character	UTF-8	UTF-16 or UCS-2
Half-width %	X’25’	X’0025’
Full-width %	X’EFBC85’	X’FF05’
Half-width_	X’5F’	X’005F’
Full-width_	X’EFBCBF’	X’FF3F’

The full-width or half-width % matches zero or more characters. The full-width or half width _ character matches exactly one character. (For ASCII or EBCDIC data, a full-width _ character matches one DBCS character.)

35. In DB2 UDB for iSeries, redundant shifts are normally ignored, but the IGNORE_LIKE_REDUNDANT_SHIFTS option must be specified to ensure redundant shifts are always ignored.

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 the 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 the search was intended to search only for the values that start with 'WYSE', then the value 'WSYE%%%%%%' should be assigned to the host variable.

ESCAPE *escape-expression*

The ESCAPE clause allows the definition of patterns intended to match values that contain the actual percent and underscore characters. The following rules govern the use of the ESCAPE clause:

- If a character-string column is identified, the escape character must be a character-string constant or fixed length variable of length 1. [36]
- If a graphic-string column is identified, the escape character must be a graphic-string constant or fixed length variable of length 1. [37]
- If the ESCAPE *host-variable* has a negative indicator variable, the result of the predicate is unknown.
- The *host-variable* or *string-constant* forming the pattern 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 is an error.
- If the column is mixed data, the ESCAPE clause is not allowed.

The following example shows the effect of successive occurrences of the escape character, which in this case is the plus sign (+).

Table 17.

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

36. Except in C, where a NUL-terminated character string variable of length 2 can be used.

37. Except in C, where a NUL-terminated graphic string variable of length 2 can be used.

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.

```
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: 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:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaa $^{S}_{O}$ AB%C $^{S}_{I}$'	'aaa $^{S}_{O}$ ABDZC $^{S}_{I}$'	True
	'aaa $^{S}_{O}$ AB $^{S}_{I}$ dzx $^{S}_{O}$ C $^{S}_{I}$'	True
	Empty string	False
WHERE COL1 LIKE 'aaa $^{S}_{O}$ AB $^{S}_{I}$ % $^{S}_{O}$ C $^{S}_{I}$'	'aaa $^{S}_{O}$ ABDZC $^{S}_{I}$'	True
	'aaa $^{S}_{O}$ AB $^{S}_{I}$ dzx $^{S}_{O}$ C $^{S}_{I}$'	True
	Empty string	False
WHERE COL1 LIKE '$^{S}_{O}$ $^{S}_{I}$'	'aaa $^{S}_{O}$ ABDZC $^{S}_{I}$'	False
	'aaa $^{S}_{O}$ AB $^{S}_{I}$ dzx $^{S}_{O}$ C $^{S}_{I}$'	False
	Empty string	True
WHERE COL1 LIKE '$^{S}_{O}$ % $^{S}_{I}$'	'aaa $^{S}_{O}$ ABDZC $^{S}_{I}$'	True
	'aaa $^{S}_{O}$ AB $^{S}_{I}$ dzx $^{S}_{O}$ C $^{S}_{I}$'	True
	Empty string	False
WHERE COL1 LIKE '$^{S}_{O}$ ___ $^{S}_{I}$ %'	'aaa $^{S}_{O}$ ABDZC $^{S}_{I}$'	True
	'aaa $^{S}_{O}$ AB $^{S}_{I}$ dzx $^{S}_{O}$ C $^{S}_{I}$'	True
	Empty string	False
WHERE COL1 LIKE '$^{S}_{O}$ ___ $^{S}_{I}$'	'aaa $^{S}_{O}$ ABDZC $^{S}_{I}$'	False
	'aaa $^{S}_{O}$ AB $^{S}_{I}$ dzx $^{S}_{O}$ C $^{S}_{I}$'	False
	Empty string	False

Example 5: In the following table of ASCII 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:

Predicates	COL1 Values	Result
WHERE COL1 LIKE 'aaaAB%C'	'aaaABDZC'	True
WHERE COL1 LIKE 'aaaAB%C'	'aaaABdzxC'	True

NULL Predicate

The NULL predicate tests for null values.

The result of a NULL predicate cannot be unknown. If the value of the column is null, the result is true. If the value is not null, the result is false.

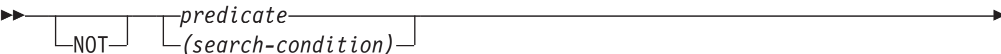
If NOT is specified, the result is reversed.

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 18. Truth Tables for AND and OR

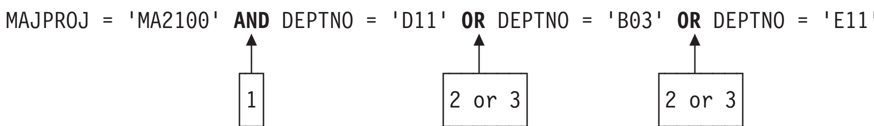
P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

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.

Examples

In the examples, the numbers on the second line indicate the order in which the operators are evaluated.



```
MAJPROJ = 'MA2100' AND (DEPTNO = 'D11' OR DEPTNO = 'B03') OR DEPTNO = 'E11'
                    ↑                  ↑                 ↑
                   [2]                [1]               [3]
```

Chapter 3. Built-in Functions

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the *built-in functions* listed in the following tables. For more information on functions, see “Functions” on page 91.

Table 19. Column Functions

Function	Description	Reference
AVG	Returns the average of a set of numbers	134
COUNT	Returns the number of rows or values in a set of rows or values	135
COUNT_BIG	Returns the number of rows or values in a set of rows or values (COUNT_BIG is similar to COUNT except that the result can be greater than the maximum value of integer)	136
MAX	Returns the maximum value in a set of values in a group	137
MIN	Returns the minimum value in a set of values in a group	138
STDDEV	Returns the biased standard deviation of a set of numbers	139
SUM	Returns the sum of a set of numbers	140
VARIANCE or VAR	Returns the biased variance of a set of numbers	141

Table 20. Cast Scalar Functions

Function	Description	Reference
BLOB	Returns a BLOB representation of a string of any type	148
CHAR	Returns a CHARACTER representation of a value	150
CLOB	Returns a CLOB representation of a value	155
DATE	Returns a DATE from a value	159
DBCLOB	Returns a DBCLOB representation of a string	166
DECIMAL	Returns a DECIMAL representation of a number	167
DOUBLE_PRECISION or DOUBLE	Returns a DOUBLE PRECISION representation of a number	171
FLOAT	Returns a FLOAT representation of a number	174
GRAPHIC	Returns a GRAPHIC representation of a string	176
INTEGER or INT	Returns an INTEGER representation of a number	179
REAL	Returns a REAL representation of a number	202
SMALLINT	Returns a SMALLINT representation of a number	209
TIME	Returns a TIME from a value	215

© Copyright IBM Corp. 1982, 2002

Table 20. Cast Scalar Functions (continued)

Function	Description	Reference
TIMESTAMP	Returns a TIMESTAMP from a value or a pair of values	216
VARCHAR	Returns a VARCHAR representative of a value	225
VARGRAPHIC	Returns a VARGRAPHIC representation of a value	227

Table 21. Datetime Scalar Functions

Function	Description	Reference
DAY	Returns the day part of a value	161
DAYOFWEEK	Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday	162
DAYOFWEEK_ISO	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday	163
DAYOFYEAR	Returns the day of the year from a value	164
DAYS	Returns an integer representation of a date	165
HOUR	Returns the hour part of a value	178
JULIAN_DAY	Returns an integer value representing a number of days from January 1, 4712 B.C. to the date specified in the argument	180
MICROSECOND	Returns the microsecond part of a value	190
MIDNIGHT_SECONDS	Returns an integer value representing the number of seconds between midnight and a specified time value	191
MINUTE	Returns the minute part of a value	192
MONTH	Returns the month part of a value	194
QUARTER	Returns an integer that represents the quarter of the year in which a date resides	199
SECOND	Returns the seconds part of a value	206
WEEK	Returns the week of the year from a value, where the week starts with Sunday	230
WEEK_ISO	Returns the week of the year from a value, where the week starts with Monday	231
YEAR	Returns the year part of a value	232

Table 22. Miscellaneous Scalar Functions

Function	Description	Reference
COALESCE	Returns the first argument that is not null	156
HEX	Returns a hexadecimal representation of a value	177
LENGTH	Returns the length of a value	183
NULLIF	Returns a null value if the arguments are equal, otherwise it returns the value of the first argument	195
VALUE	Returns the first argument that is not null	224

Table 23. Numeric Scalar Functions

Function	Description	Reference
ABS	Returns the absolute value of a number	143
ACOS	Returns the arc cosine of a number, in radians	144
ASIN	Returns the arc sine of a number, in radians	145
ATAN	Returns the arc tangent of a number, in radians	146
ATAN2	Returns the arc tangent of x and y coordinates as an angle expressed in radians	147
CEILING	Returns the smallest integer value that is greater than or equal to a number	149
COS	Returns the cosine of a number	158
DEGREE	Returns the number of degrees of an angle	169
DIGITS	Returns a character-string representation of the absolute value of a number	170
EXP	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument	173
FLOOR	Returns the largest integer value that is less than or equal to a number	175
LN	Returns the natural logarithm of a number	184
LOG10	Returns the common logarithm (base 10) of a number	187
MOD	Returns the remainder of the first argument divided by the second argument	193
POWER	Returns the result of raising the first argument to the power of the second argument	198
RADIANS	Returns the number of radians for an argument that is expressed in degrees	200
RAND	Returns a random number	201
ROUND	Returns a numeric value that has been rounded to the specified number of decimal places	203
SIGN	Returns the sign of a number	207
SIN	Returns the sine of a number	208
SQRT	Returns the square root of a number	211
TAN	Returns the tangent of a number	214
TRUNCATE or TRUNC	Returns a number value that has been truncated at a specified number of decimal places	220

Table 24. String Scalar Functions

Function	Description	Reference
CONCAT	Returns a string that is the concatenation of two strings	157
LCASE	Returns a string in which all the characters have been converted to lowercase characters	181
LEFT	Returns the leftmost characters from the string	182

Table 24. String Scalar Functions (continued)

Function	Description	Reference
LOCATE	Returns the starting position of one string within another string	185
LOWER	Returns a string in which all the characters have been converted to lowercase characters	188
LTRIM	Returns a string in which blanks have been removed from the beginning of another string	189
POSSTR	Returns the starting position of one string within another string	196
RTRIM	Returns a string in which blanks have been removed from the end of another string	205
SPACE	Returns a character string that consists of a specified number of blanks	210
SUBSTR	Returns a substring of a string	212
TRANSLATE	Returns a string in which one or more characters in a string are converted to other characters	218
UCASE	Returns a string in which all the characters have been converted to uppercase characters	222
UPPER	Returns a string in which all the characters have been converted to uppercase characters	223

Column Functions

The following information applies to all column functions. However, it does not apply when an asterisk (*) is used as the argument to COUNT or COUNT_BIG.

- The argument of a column function is a set of values derived from an expression. The expression must include a column name and must not include another column function. The scope of the set is a group or an intermediate result table as explained in Chapter 4, "Queries" on page 233.
- 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.
- 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 applied to the empty set because department D01 has no employees:

```
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, redundant duplicate values are eliminated. If ALL is implicitly or explicitly specified, redundant 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 can be of 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 double-precision floating point if the argument values are single-precision floating point.
- The result is a 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 numbers with precision *p* and scale *s*.[38] In DB2 UDB for z/OS and OS/390, the scale is MAX(0,28-*p*+*s*).

The function is applied to the set of values derived from the argument values by eliminating null values. If DISTINCT is specified, redundant duplicate values are eliminated.

The result can be null. If the 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'
  ```

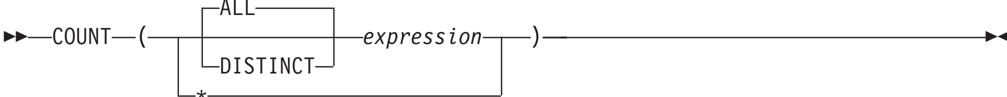
  Results in AVERAGE being set to 4.25 (that is, 17/4).
- Using the PROJECT table, set the host variable ANY_CALC to the average of each unique staffing level value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

  ```
  SELECT AVG(DISTINCT PRSTAFF)
    INTO :ANY_CALC
    FROM PROJECT
    WHERE DEPTNO = 'D11'
  ```

  Results in ANY_CALC being set to 4.66 (that is, 14/3).

38. For DB2 UDB for z/OS and OS/390, the formulas used in this book are those that apply when the DEC31 option is in effect or the precision of an operand is greater than 15.

COUNT



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

The argument values can be of any built-in data type other than a BLOB, CLOB, or DBCLOB. If DISTINCT is used, the resulting *expression* must not have a length attribute greater than 255 for a character column or 127 for a graphic column.

The result of the function is a large integer and must be within the range of large integers. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set including duplicates.

The argument of COUNT(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values and redundant duplicate values. The result is the number of different non-null values in the set.

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'
```

  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

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 argument values can be of any built-in data type other than a BLOB, CLOB, or DBCLOB. If DISTINCT is used, the resulting *expression* must not have a length attribute greater than 255 for a character column or 127 for a graphic column.

The result of the function is a decimal with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT_BIG(*expression*) or COUNT_BIG(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set including duplicates.

The argument of COUNT_BIG(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values and redundant duplicate values. The result is the number of different non-null values in the set.

Examples

- Refer to the COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- To create a sourced function that is similar to the built-in COUNT_BIG function, the definition of the sourced function must include the type of the column that can be specified when the new function is invoked. In this example, the CREATE FUNCTION statement creates a sourced function that takes 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 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 the database manager locates the COUNT_BIG function.

MAX

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

The argument values can be of any built-in data type other than a BLOB, CLOB, or DBCLOB. The *expression* must not have a length attribute greater than 255 for a character column or 127 for a graphic column.

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 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 set of values is empty, the result is the null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

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 PLANNINGƀƀƀƀƀƀ'.

MIN

The MIN function returns the minimum value in a set of values.

The argument values can be of any built-in data type other than a BLOB, CLOB, or DBCLOB. The *expression* must not have a length attribute greater than 255 for a character column or 127 for a graphic column.

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 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 set of values is empty, the result is the null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and is not advised.

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

The STDDEV function returns the biased population standard deviation (/n) of a set of numbers. The formula used to calculate STDDEV is logically equivalent to:

```
STDDEV = SQRT(VAR)
```

where SQRT(VAR) is the square root of the variance.

The argument values must be of 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, redundant duplicate values are eliminated.

The result can be null. If the set of values is empty, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

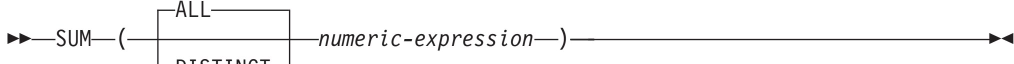
Example

- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries for those employees in department A00.

  ```
  SELECT STDDEV(SALARY)
    INTO :DEV
    FROM EMPLOYEE
    WHERE WORKDEPT = 'A00';
  ```

  Results in DEV being set to approximately 9742.43.

SUM



The SUM function returns the sum of a set of numbers.

The argument values can be of 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 numbers with precision *p* and scale *s*.[39] In DB2 UDB for z/OS and OS/390, the precision of the result is min(31, *p*+10).

G
G

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

The result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is the sum of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example

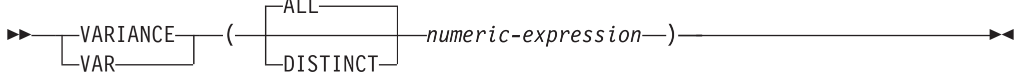
- Using the EMPLOYEE table, set the host variable JOB_BONUS (DECIMAL(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```
SELECT SUM(BONUS)
  INTO :JOB_BONUS
  FROM EMPLOYEE
  WHERE JOB = 'CLERK'
```

  Results in JOB_BONUS being set to 2800.

39. For DB2 UDB for z/OS and OS/390, the formulas used in this book are those that apply when the DEC31 option is in effect or the precision of an operand is greater than 15.

VARIANCE or VAR



The VARIANCE functions return the biased population variance (/n) of a set of numbers. The formula used to calculate VARIANCE is logically equivalent to:

```
VARIANCE = SUM(X**2)/COUNT(X) - (SUM(X)/COUNT(X))**2
```

The argument values can be of 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, redundant duplicate values are eliminated.

The result can be null. If the set of values is empty, 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 aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example

- Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the variance of the salaries for those employees in department A00.

  ```
  SELECT VARIANCE(SALARY)
    INTO :VARNCE
    FROM EMPLOYEE
    WHERE WORKDEPT = 'A00';
  ```

  Results in VARNCE being set to approximately 98 763 888.88.

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 set of parameter values rather than to sets of values. The argument of a scalar function can be a function. However, the restrictions that apply to the use of expressions and 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. In DB2 UDB for iSeries the result is an 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 π.

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.

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 $-\pi/2$ and less than or equal to $\pi/2$.

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.

ATAN2

```
►►──ATAN2──(──numeric-expression-1──,──numeric-expression-2──)──────────────────►◄
```

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 must be 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 1.107 148 7.

BLOB

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, or binary string.

integer
: Specifies the length attribute for the resulting binary string. The value must be between 1 and 2 147 483 647. For more information, see Table 39 on page 510.

  If *integer* is not specified, 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. If *integer* is not specified, the *string-expression* must not be the empty string constant.

  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.

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 109.

Examples

- The following function returns a BLOB for the string ’This is a BLOB’.

```
SELECT BLOB('This is a BLOB')
   FROM SYSIBM.SYSDUMMY1
```

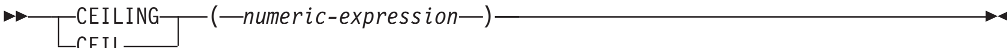
- The following function returns a BLOB for the large object that is identified by locator myclob_locator.

```
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.

```
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. For example, an
G argument with a data type of DECIMAL(5,5) will result in DECIMAL(5,0). In DB2
G UDB for UWO the result of the function is DOUBLE if the argument is REAL or
G DECIMAL.

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 000004396. 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
  ```

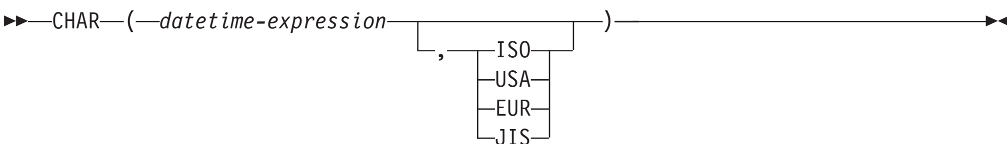
  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

  ```
  04.   04.  -03.  -03.
  ```

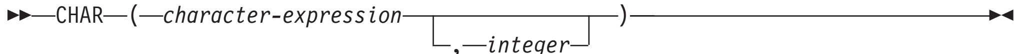
  respectively.

CHAR

Datetime to Character

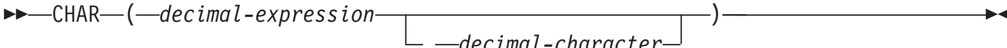


Character to Character



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 or INTEGER.
- 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.

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. The length of the result is 10. For more information see "String Representations of Datetime Values" on page 49.

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 from the default time format. The length of the result is 8. For more information see "String Representations of Datetime Values" on page 49.

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 49.

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 254. In EBCDIC environments, if the first argument is mixed data, the second argument cannot be less than 4.

If the second argument is not specified the length attribute of the result is the same as the length attribute of the first argument. The *character-expression* must not be the empty string constant.

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.

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 built-in integer data type (either SMALLINT or INTEGER).

The result is the fixed-length character-string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. The result is left justified.

- 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.

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 or NUMERIC data type. 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 75.

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.
G Leading zeros are not returned. Trailing zeros are returned. In DB2 UDB for z/OS
G and OS/390 and DB2 UDB for UWO leading zeroes are returned. In DB2 UDB for
G z/OS and OS/390, a leading blank is returned from the CHAR function for
G positive decimal values. The leading blank is not returned for
G CAST(*decimal-expression* AS CHAR(n)).

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 at least 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).

The single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number is the default decimal point. For more information, see “Decimal Point” on page 75.

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 attribute is specified; the CAST specification should be used for maximal portability. For more information, see "CAST Specification" on page 109.

Examples

- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25.

  ```
  SELECT CHAR(PRSTDATE, USA)
     FROM PROJECT
  ```

  Results in the value '12/25/1988'.
- Assume the column STARTING has an internal value equivalent to 17:12:30, the host variable HOUR_DUR (DECIMAL(6,0)) is a time duration with a value of 050000 (that is, 5 hours).

  ```
  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 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
  ```

  An EDLEVEL of 18 would be returned as the CHAR(6) value '18ƀƀƀƀ' (blank padded on the right with 4 blanks).
- Assume the same SALARY column subtracted from 20000.25 is to be returned with a comma as the decimal character.

  ```
  SELECT CHAR(20000.25 - SALARY, ',')
     FROM EMPLOYEE
  ```

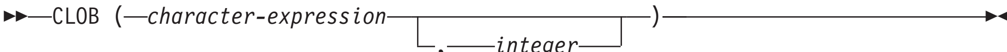
  returns the value '-1642,75ƀƀƀ' (blank padded on the right with 3 blanks).
- Assume a host variable, DOUBLE_NUM, has a double-precision floating-point data type and a value of -987.654321E-35.

  ```
  SELECT CHAR(:DOUBLE_NUM)
     FROM SYSIBM.SYSDUMMY1
  ```

Results in the character value '-9.8765432100000002E-33 '.[40]

40. Note that since floating-point numbers are approximate, the resulting character string will vary slightly based on that approximation.

CLOB



The CLOB function returns a CLOB representation of a character string.

character-expression
An expression that returns a value that is a built-in CHAR, VARCHAR, or CLOB data type. The argument must not be bit data.

integer
Specifies the length attribute for the resulting varying-length character string. The value must be between 1 and 2 147 483 647. In EBCDIC environments, if the first argument is mixed data, the second argument cannot be less than 4.

If *integer* is not specified the length attribute of the result is the same as the length attribute of the first argument. The *character-expression* must not be the empty string constant.

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.

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

The CCSID of the result is the same as the CCSID of the first 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 109.

Example

- The following function returns a CLOB for the string ’This is a CLOB’.

```
SELECT CLOB('This is a CLOB')
  FROM SYSIBM.SYSDUMMY1
```

COALESCE

The COALESCE function returns the value of the first non-null expression.

The arguments must be compatible. For more information on data type compatibility, see “Assignments and Comparisons” on page 58. The arguments can be of either a built-in or user-defined data type.[41]

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 68.

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
```

41. 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 on data type compatibility, see “Assignments and Comparisons” on page 58.

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 98.

Example

- Concatenate the column FIRSTNME with the column LASTNAME.

```
SELECT CONCAT(FIRSTNME, 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.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.
  - 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.
- If *expression* is a number, it must be greater than or equal to one and 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 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 109.

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
  WHERE SOURCE = 'BADAMSON'
```

  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

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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.

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 163.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.

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(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—(—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 162.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.

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'.

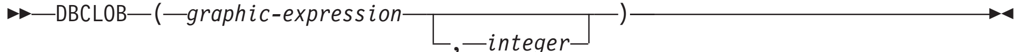
  ```
  SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
    INTO :EDUCATION_DAYS
    FROM PROJECT
    WHERE PROJNO = 'IF2000'
  ```

  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 1584.

DBCLOB



The DBCLOB function returns a DBCLOB representation of a graphic string expression.

graphic-expression
: An expression that returns a value that is a GRAPHIC or VARGRAPHIC or DBCLOB data type.

integer
: Specifies the length attribute for the resulting varying-length graphic string. The value must be between 1 and 1 073 741 823.

  If *integer* is not specified the length attribute of the result is the same as the length attribute of the first argument. The *graphic-expression* must not be the empty string constant.

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.

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.

The CCSID of the result is the same as the CCSID of the first 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 109.

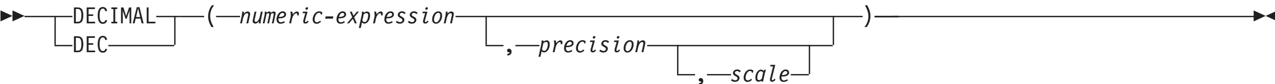
Example

- 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'.

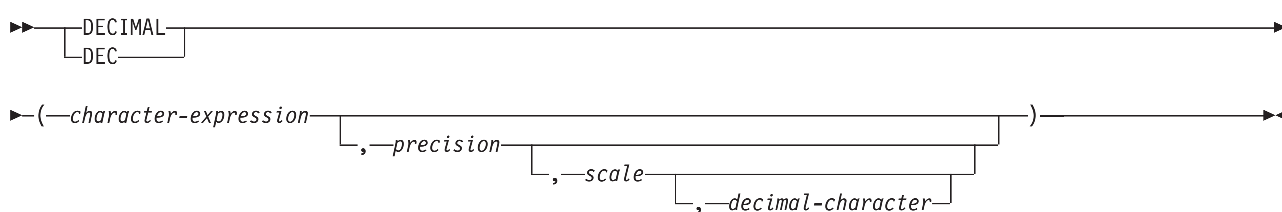
```
SELECT DBCLOB(FIRSTNME)
  INTO :VAR_DESC
  FROM EMPLOYEE
  WHERE EMPNO = '000050'
```

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

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 or decimal
- 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 is returned 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. 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 and must have an actual length that is not greater than 255 bytes.

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 75.

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 is returned 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 a different *decimal-character* 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 109.

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 21 400.50.

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.

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, DECIMAL, or NUMERIC.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal point. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- *p* if the argument is a decimal number with a precision of *p*.

The CCSID of the character string is the default 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. Then, for this value:

  ```
  SELECT DIGITS(COLUMNX)
    FROM TABLEX
  ```

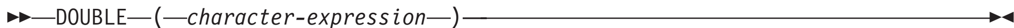
  returns the value '000628'.

  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.

DOUBLE_PRECISION or DOUBLE

Numeric to Double

Character to Double



The DOUBLE_PRECISION and DOUBLE functions return a 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 Double

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the expression were assigned to a double-precision floating-point column or variable.

Character to Double

character-expression
An expression that returns a character string value. The argument must not be a CLOB and must have an actual length that is not greater than 255 bytes.

The result is the same number that would result from CAST(*character-expression* AS DOUBLE PRECISION). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an floating-point, integer, or decimal constant.

The single-byte character constant that must be used to delimit the decimal digits in *character-expression* from the whole part of the number is the default decimal point. For more information, see "Decimal Point" on page 75.

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Note

Syntax alternatives: FLOAT is a synonym for DOUBLE_PRECISION and DOUBLE.

The CAST specification should be used for maximal portability. For more information, see "CAST Specification" on page 109.

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

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

```
►►—FLOAT—(—numeric-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 171.

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
G DECIMAL(5,0). In DB2 UDB for UWO the result of the function is DOUBLE if the
G argument is REAL or DECIMAL.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Use the FLOOR function to truncate any digits to the right of the decimal point.

  ```
  SELECT FLOOR(SALARY)
    FROM EMPLOYEE
    WHERE LASTNAME = 'HAAS'
  ```

  This example returns 52 750.
- Use FLOOR on both positive and negative numbers.

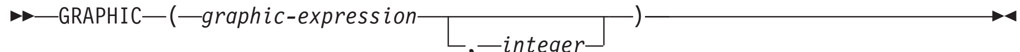
  ```
  SELECT FLOOR( 3.5),
         FLOOR( 3.1),
         FLOOR(-3.1),
         FLOOR(-3.5)
    FROM SYSIBM.SYSDUMMY1
  ```

  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

  ```
  03.   03.  -04.  -04.
  ```

  respectively.

GRAPHIC



The GRAPHIC function returns a fixed-length graphic-string representation of a graphic-string expression.

graphic-expression
: Specifies a graphic-string expression.

integer
: Specifies the length attribute of the result and must be an integer constant between 1 and 127. If the length of *graphic-expression* is less than *integer*, the result is padded with double-byte blanks to the length of the result.

  If *integer* is not specified, the length of the result is the minimum of 127 and the length attribute of *graphic-expression*.

If the length of the *graphic-expression* is greater than the length attribute of the result, truncation is performed.

The result of the function is a fixed-length graphic string (GRAPHIC).

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 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 109.

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'
```

HEX

The HEX function returns a hexadecimal representation of a value.

The argument must be an expression that returns a value of any built-in data type other than a character or binary string with a length attribute greater than 255 or a graphic string with a length attribute greater than 127.

The result of the function is a character string. If the argument can be null, the result can be null; if the argument is null, the result is a 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.

If the argument is not a graphic string, the length of the result is twice the length of the argument. If the argument is a graphic string, the length of the result is four times the length of the argument.

If the argument is a fixed-length string and the length of the result is less than the product-specific maximum length attribute of CHAR (or GRAPHIC), the result is a fixed-length string. For more information on the product-specific maximum length, see Appendix A, “SQL Limits” on page 509. Otherwise, the result is a varying-length string whose length attribute depends on the following:

- If the argument is a character or binary string, the length attribute of the result is twice the length attribute of the argument.
- If the argument is a graphic string, the length attribute of the result is four times the length attribute of the argument.

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 FIRSTNME, MIDINIT, LASTNAME, HEX(EDLEVEL)
  FROM EMPLOYEE
```

HOUR

```
►►—HOUR—(—expression—)——————————————————————————————————►◄
```

The HOUR 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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.

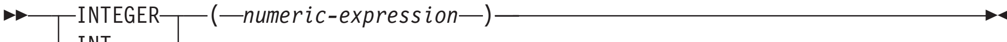
Example

- Using the CL_SCHED sample table, select all the classes that start in the afternoon.

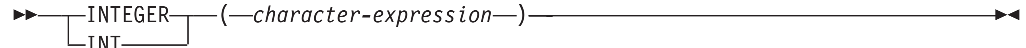
```
SELECT *
  FROM CL_SCHED
  WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

INTEGER or INT

Numeric to Integer



Character to Integer



The INTEGER function returns an integer representation of:

- A number
- A character-string representation of an integer

Numeric to Integer

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 large integer column or variable. If the whole part of the argument is not within the range of integers, an error is returned. 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 and must have an actual length that is not greater than 255 bytes.

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 an integer constant. If the whole part of the argument is not within the range of integers, an error is returned.

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

Note

Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see "CAST Specification" on page 109.

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 INTEGER(SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
  FROM EMPLOYEE
```

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 and must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.

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 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 2 438 762.
- 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 2 450 815.

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 188.

LEFT

```
►►—LEFT—(—string-expression—,—integer—)——————————————————►◄
```

The LEFT function returns the leftmost *integer* bytes of *string-expression*.

string-expression
: An expression that specifies the string from which the result is derived. *string-expression* must be a character string or a binary string with a built-in data type. If *string-expression* is a BLOB or CLOB, it must not have a length attribute greater than 1M. Otherwise, *string-expression* must not have a length attribute greater than 4000. A substring of *string-expression* is zero or more contiguous bytes of *string-expression*.[42]

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*. It must not, however, be the integer constant 0.

  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*:

- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- BLOB if *string-expression* is BLOB

G In DB2 UDB for iSeries if *integer* is an integer constant and the argument is not a
G BLOB, CLOB, or DBCLOB, the result of the function is a fixed-length string. In
G DB2 UDB for UWO the length attribute is 4000 if the data type is VARCHAR and
G 1M if the data type is CLOB or BLOB.

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*.

Example

- Assume the host variable NAME (VARCHAR(50)) has a value of 'KATIE AUSTIN' and the host variable FIRSTNAME_LEN (INTEGER) has a value of 5.

  ```
  SELECT LEFT(:NAME, :FIRSTNAME_LEN)
     FROM SYSIBM.SYSDUMMY1
  ```

  Returns the value 'KATIE'

42. In EBCDIC environments, if the *string-expression* contains mixed data, the LEFT function operates on a strict byte-count basis. Because LEFT operates on a strict byte-count basis, the result is not necessarily a properly formed mixed data character string.

LENGTH

```
►►──LENGTH──(──expression──)──────────────────────────────────────────►◄
```

The LENGTH function returns the length of a value.

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
- The integer 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
- 4 for date
- 3 for time
- 10 for timestamp

Examples

- Assume the host variable ADDRESS is a varying-length character string with a value of '895 Don Mills Road'.

  ```
  SELECT LENGTH(:ADDRESS)
     FROM SYSIBM.SYSDUMMY1
  ```

  Returns the value 18.
- Assume that PRSTDATE is a column of type DATE.

  ```
  SELECT LENGTH(PRSTDATE)
     FROM PROJECT
     WHERE PROJNO = 'AD3111'
  ```

  Returns the value 4.
- Assume that PRSTDATE is a column of type DATE.

  ```
  SELECT LENGTH(CHAR(PRSTDATE, EUR))
     FROM PROJECT
     WHERE PROJNO = 'AD3111'
  ```

  Returns the value 10.

LN

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.

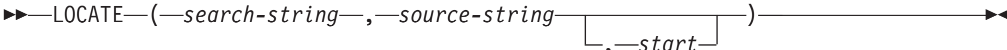
Example

- 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.

LOCATE



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. The search string must be a character or binary string with an actual length that is no greater than 4000 bytes. It must be compatible with the *source-string*. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable
- A scalar function whose arguments are a constant, a special register, or a host variable (though nested function invocations cannot be used)
- An expression that concatenates any of the above

source-string
: An expression that specifies the source string in which the search is to take place. The source string must be a character or binary string with an actual length that is no greater than 4000 bytes. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable
- A scalar function whose arguments are a constant, a special register, or a host variable (though nested function invocations cannot be used)
- A column name
- An expression that concatenates any of the above

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 “POSSTR” on page 196.

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 argument value must 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.

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.

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 or Unicode 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.

string-expression
An expression that specifies the string to be converted. *String-expression* must be a character string or Unicode graphic string. An argument with a character string data type must not be a CLOB and must have an actual length that is not greater than 255 bytes. An argument with a graphic string data type must not be a DBCLOB and must have an actual length that is not greater than 127 bytes.

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.

Example

- 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 from the beginning of a string expression.

The argument must be any character string expression other than a CLOB or any graphic string expression other than a DBCLOB. The characters that are interpreted as leading blanks depend on the encoding scheme of the data and the data type:

- If the argument is a DBCS graphic string, then the leading DBCS blanks are removed.
- If the first argument is a Unicode graphic string, then the leading Unicode blanks are removed
- Otherwise, leading SBCS blanks are removed.

The data type of the result depends on the data type of *string-expression*:

Data type of ***expression***	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

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 blanks 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’.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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 999 999.
- If the argument is a duration:

  The result is the microsecond part of the value, which is an integer between –999 999 and 999 999. 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 and must have an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.

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

- 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 47 410. 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 47 410 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 86 400 and 0. Although these two values represent the same point in time, different values are returned.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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:

```
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 be each be an expression that returns a built-in INTEGER or SMALLINT 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.

G G The result of the function is a large integer. In DB2 UDB for UWO if both arguments are small integers, the result is a small integer.

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.

MONTH

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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
```

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 built-in data types. Neither argument can be a BLOB, CLOB, or DBCLOB. Character-string arguments are compatible and comparable with datetime 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.

POSSTR

```
►►—POSSTR—(—source-string—,—search-string—)——————————————————————►◄
```

The POSSTR functions return 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*. See the related function, “LOCATE” on page 185.

source-string
: An expression that specifies the source string in which the search is to take place. *source-string* must be a character, graphic, or binary string expression. The expression can be specified by any of the following:
  - A constant
  - A special register
  - A host variable (including a LOB locator variable)
  - A scalar function whose arguments are a constant, a special register, or a host variable (though nested function invocations cannot be used)
  - A column name
  - An expression that concatenates any of the above

search-string
: An expression that specifies the string that is to be searched for. *search-string* must be a character or graphic string that is not a LOB with an actual length that is no greater than 4000 bytes. It must be compatible with the *source-string*. The expression can be specified by any of the following:
  - A constant
  - A special register
  - A host variable
  - A scalar function whose arguments are a constant, a special register, or a host variable (though nested function invocations cannot be used)
  - An expression that concatenates any of the above

The result of the function is a large integer. If either of the arguments can be null, the result can be null. If either of the arguments is null, the result is the null value.

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis without regard to single-byte or double-byte characters.[43]

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*.

If the *search-string* has a length of zero, the result returned by the function is 1. Otherwise:

- if the *source-string* has a length of zero, the result returned by the function is 0.
- Otherwise,

43. For example, in EBCDIC environments, if the *source-string* contains mixed data, the *search-string* will only be found if any shift-in and shift-out characters are also found in the *source-string* in exactly the same positions.

– If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, then the result returned by the function is the starting position of the first such substring within the *source-string* value.
– Otherwise, the result returned by the function is 0.[44]

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
```

44. 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. [45]

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.

G
G
G

The result of the function is a double-precision floating-point number. In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO the result is INTEGER if both arguments are either INTEGER or SMALLINT. 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.

45. The result of the POWER function is exactly the same as the result of exponentiation: *numeric-expression-1* ** *numeric-expression-2*.

QUARTER

The QUARTER function returns an integer between 1 and 4 that represents the quarter of the year in which the date resides. For example, any dates in January, February, or March 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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.

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.

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.

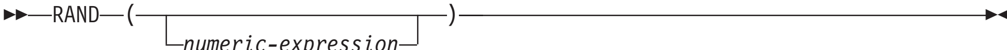
Example

- 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 between 0 and 2 147 483 646.

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.

A specific seed value will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed.

RAND is a non-deterministic function.

Examples

- 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──)─────────────────────────────────────▶◀
```

The REAL function returns a single-precision floating-point representation of a number.

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable. If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned.

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 109.

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──(──numeric-expression-1──,──numeric-expression-2──)──────────────────►◄
```

The ROUND function returns *numeric-expression–1* rounded to *numeric-expression–2* 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 1 + (the absolute value of *numeric-expression–2*) 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
  ```

  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

  ```
  0873.730   0873.700  0874.000  0870.000  0900.000  1000.000  0000.000
  ```

  respectively.
- Calculate both positive and negative numbers.

```
SELECT ROUND( 3.5, 0),
       ROUND( 3.1, 0),
       ROUND(-3.1, 0),
       ROUND(-3.5, 0)
  FROM TABLEX
```

This example returns:

```
04.0   03.0  -03.0  -04.0
```

respectively.

RTRIM

```
►►──RTRIM──(──string-expression──)─────────────────────────────────►◄
```

The RTRIM function removes blanks from the end of a string expression.

The argument must be any character string expression other than a CLOB or any graphic string expression other than a DBCLOB. The characters that are interpreted as trailing blanks depend on the encoding scheme of the data and the data type:

- If the argument is a DBCS graphic string, then the trailing DBCS blanks are removed.
- If the first argument is a Unicode graphic string, then the trailing Unicode 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 *string-expression*	**Data type of the Result**
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

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 blanks 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’.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of times and timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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
     WHERE SOURCE = 'BADAMSON'
  ```

  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, except DECIMAL(31,31).

The result has the same data type and length attribute as the argument, except that precision is increased by one if the argument is DECIMAL or NUMERIC and the scale of the argument is equal to its precision. For example, an argument with a
G data type of DECIMAL(5,5) will result in DECIMAL(6,5). In DB2 UDB for UWO
G the result is DOUBLE if the argument is DECIMAL or REAL.

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 50 000.

```
SELECT SIGN(:PROFIT)
  FROM SYSIBM.SYSDUMMY1
```

This example returns the value 1.

SIN

The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. 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.

SMALLINT

Numeric to Smallint

```
►►──SMALLINT──(──numeric-expression──)─────────────────────────────────►◄
```

Character to Smallint

```
►►──SMALLINT──(──character-expression──)───────────────────────────────►◄
```

The SMALLINT function returns an integer representation of:

- A number
- A character-string representation of an integer

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 is returned. 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 and must have an actual length that is not greater than 255 bytes.

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 an integer constant. If the whole part of the argument is not within the range of small integers, an error is returned.

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

Note

Syntax alternatives: The CAST specification should be used for maximal portability. For more information, see "CAST Specification" on page 109.

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
```

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 4000. 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.
G Otherwise, the length attribute of the result is 4000. In DB2 UDB for UWO the
G length attribute is always VARCHAR(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 of the result is the default CCSID at the current server.

Example

- The following statement returns a character string that consists of 5 blanks.

```
SELECT SPACE(5)
  FROM SYSIBM.SYSDUMMY1
```

SQRT

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.

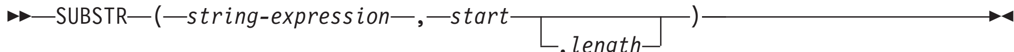
Example

- 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.

SUBSTR



The SUBSTR function returns 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 Unicode 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 for character strings, 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 an integer that is greater than zero and less than or equal to 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 *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 *string-expression* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*string-expression*) - *start* + 1 which is the number of characters (or bytes) from the character (or byte) specified by start to the last character (or byte) of *string-expression*. If *string-expression* is a varying-length string, omission of *length* is an implicit specification of the greater of zero or LENGTH(*string-expression*) - *start* + 1. 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*:

Data type of *string-expression*	**Data Type of the Result for SUBSTR**
CHAR or VARCHAR	CHAR, if: • *length* is explicitly specified by an integer constant that is less than the product-specific maximum of a character-string. See Table 39 on page 510 for more information. • *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant. VARCHAR, in all other cases.

Data type of *string-expression*	Data Type of the Result for SUBSTR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	GRAPHIC, if: • *length* is explicitly specified by an integer constant that is less than the product-specific maximum of a graphic-string. See Table 39 on page 510 for more information. • *length* is not explicitly specified, but *string-expression* is a fixed-length string and *start* is an integer constant. VARGRAPHIC, in all other cases.
DBCLOB	DBCLOB
BLOB	BLOB

If *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 LENGTH(*string-expression*) - *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'

  ```
  SELECT SUBSTR(:NAME, :SURNAME_POS, 1)
     FROM SYSIBM.SYSDUMMY1
  ```

  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.

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 time or timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.

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 109.

Example

- Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT *
  FROM IN_TRAY
  WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

TIMESTAMP

The 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-1* 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 timestamp with an actual length that is not greater than 255 bytes. For the valid formats of string representations of timestamps, see "String Representations of Datetime Values" on page 49.

  – A character string with an actual length of 14 that represents a valid date and time in the form yyyyxxddhhmmss, 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 *expression-1* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a date with an actual length that is not greater than 255 bytes. If *expression-2* is a character string, it must not be a CLOB and its value must be a valid character-string representation of a time with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and times, see "String Representations of Datetime Values" on page 49.

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 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 109.

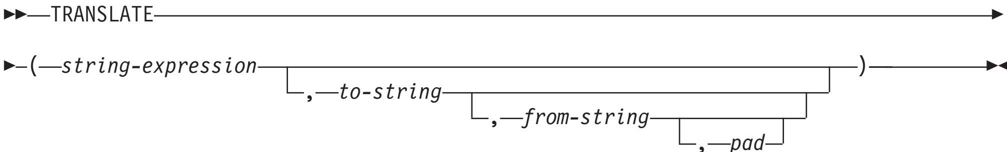
Example

- 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'.

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. A character string argument must not be a CLOB and must have an actual length that is not greater than 255.

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 not have an actual length that is greater than 255.

  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 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 greater than 255.

  If *from-string* contains duplicate characters, the left-most one is used and no warning is issued. The default value for *from-string* is a string of all bit representations starting with X'00' and ending with 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 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. For more information, see "UPPER" on page 223.

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.

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’.

- In an EBCDIC environment, monocase the mixed data character string.

```
SELECT TRANSLATE('ab SO C SI def')
```

  Returns the value 'AB SO C SI DEF'

- 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
```

- Given the same host variable SITE with a value of ’Pivabiska Lake Place’:

```
SELECT TRANSLATE(:SITE, '$$', 'Ll')
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value ’Pivabiska $ake P$ace’.

- Given the same host variable SITE with a value of ’Pivabiska Lake Place’:

```
SELECT TRANSLATE(:SITE, 'pLA', 'Place', '.')
   FROM SYSIBM.SYSDUMMY1
```

  Returns the value ’pivAbiskA LAk. pLA..’.

TRUNCATE or TRUNC

```
►►──┬─TRUNCATE─┬──(──numeric-expression-1──,──numeric-expression-2──)──────────►◄
    └─TRUNC────┘
```

The TRUNCATE function returns *numeric-expression–1* truncated to *numeric-expression–2* 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 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 1 + (the absolute value of *numeric-expression–2*) number of places to the left of the decimal point. If 1 + (the absolute value of *numeric-expression–2*) is greater than or equal to the number of digits to the left of the decimal point, the result is 0.

The data type and length attribute of the result are the same as the data type and
G length attribute of the first argument. In DB2 UDB for UWO the result is INTEGER
G if the first argument is SMALLINT or INTEGER. Otherwise, the result is DOUBLE.

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),
       TRUNCATE(873.726, -4)
  FROM SYSIBM.SYSDUMMY1
```

  This example returns (leading zeroes are shown to demonstrate the precision and scale of the result):

```
0873.720   0873.700  0873.000  0870.000  0800.000  0000.000  0000.000
```

  respectively.

- Calculate both positive and negative numbers.

```
SELECT TRUNCATE( 3.5, 0),
       TRUNCATE( 3.1, 0),
       TRUNCATE(-3.1, 0),
       TRUNCATE(-3.5, 0)
  FROM TABLEX
```

This example returns:

```
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 223.

UPPER

The UPPER function returns a string in which all the characters have been converted to uppercase characters, based on the CCSID of the argument. Only SBCS or Unicode 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.

string-expression
: An expression that specifies the string to be converted. *String-expression* must be a character or Unicode graphic string. An argument with a character string data type must not be a CLOB and must have an actual length that is not greater than 255 bytes. An argument with a graphic string data type must not be a DBCLOB and must have an actual length that is not greater than 127 bytes.

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: UCASE is a synonym for UPPER.

Examples

- Uppercase the string ’abcdef’ using the UPPER scalar function.

  ```
  SELECT UPPER('abcdef')
       FROM SYSIBM.SYSDUMMY1
  ```

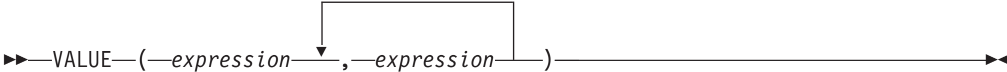
  Returns the value ’ABCDEF’.

- In EBCDIC environments, uppercase the mixed data character string using the UPPER scalar function.

  UPPER(’ab $^S{}_O$ ⊂ $^S{}_I$ def’)

  Returns the value: ’AB $^S{}_O$ ⊂ $^S{}_I$ DEF’

VALUE

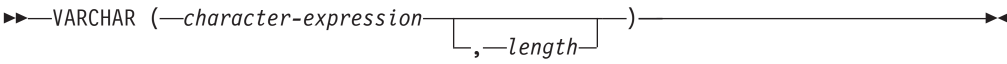


The VALUE function returns the value of the first non-null expression.

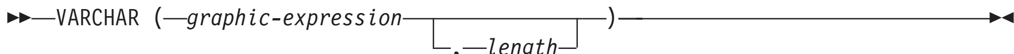
The VALUE function is identical to the COALESCE function. For more information, see “COALESCE” on page 156.

VARCHAR

Character to Varchar



Graphic to Varchar



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

- A character string if the first argument is any type of character string
- A graphic string if the first argument is a Unicode 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 character string data type. The argument must not be a CLOB and must have an actual length that is not greater than 32672 bytes.

length
: Specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32672. In EBCDIC environments, 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 an empty string constant, the length attribute of the result is 1. In DB2 UDB for UWO the length attribute is 0.
  - 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.

G

The CCSID of the result is the same as the CCSID of *character-expression*.

Graphic to Varchar

graphic-expression
: The argument must be a Unicode graphic string and must not be a DBCLOB and must have an actual length that is not greater than 16336 characters.

length
: Specifies the length attribute for the resulting varying length character string. The value must be between 1 and 32672. In EBCDIC environments, if the result is mixed data, the second argument cannot be less than 4.

If the second argument is not specified, the length attribute of the result is determined as follows (where *n* is the length attribute of the first argument):

- If the *graphic-expression* is the empty graphic string constant, the length
G attribute of the result is 1. In DB2 UDB for UWO the length attribute is 0.
- If the result is SBCS data, the length attribute of the result is *n*.
- If the result is mixed data, the length attribute of the result is
G product-specific.

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.

The CCSID of the result is the default 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 109.

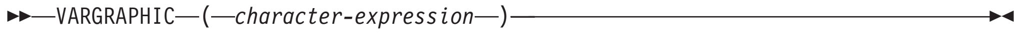
Example

- Make EMPNO varying-length with a length of 10.

```
SELECT VARCHAR(EMPNO,10)
  INTO :VARHV
  FROM EMPLOYEE
```

VARGRAPHIC

Character to Vargraphic



Graphic to Vargraphic

The VARGRAPHIC function returns a varying-length graphic-string representation of a string.

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 Vargraphic

character-expression
An expression that returns a value that is a character string. It cannot be bit data.[46]

If the expression is an empty string or the EBCDIC string `X'0E0F'`, the length attribute of the result is 1. In DB2 UDB for UWO the length attribute is 0. Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

G

The actual length of the result is the minimum of the length attribute of the result and the actual length of *character-expression*. If the length of *character-expression*, as measured in single-byte characters, is greater than the specified length of the result, as measured in double-byte characters, the result is truncated.

The CCSID of the result is determined by a mixed data CCSID. Let M denote that mixed data 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 23 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:

46. Although DB2 UDB for z/OS and OS/390 supports storing ASCII data, expression can only be in the EBCDIC encoding scheme.

– 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'
932	301	Japanese ASCII	X'FCFC'
933	834	Korean EBCDIC	X'FEFE'
934	926	Korean ASCII	X'BFFC'
935	837	S-Chinese EBCDIC	X'FEFE'
936	928	S-Chinese ASCII	X'FCFC'
937	835	T-Chinese EBCDIC	X'FEFE'
938	927	T-Chinese ASCII	X'FCFC'
939	300	Japanese EBCDIC	X'FEFE'
943	941	Japanese ASCII	X'FCFC'
942	301	Japanese ASCII	X'FCFC'
944	926	Korean ASCII	X'BFFC'
946	928	S-Chinese ASCII	X'FCFC'
948	927	T-Chinese ASCII	X'FCFC'
949	951	Korean ASCII	X'AFFE'
950	947	T-Chinese ASCII (Big-5)	X'C8FE'
954	13488	Japanese ASCII EUC	X'FFFD'
964	13488	T-Chinese ASCII EUC	X'FFFD'
1363	1362	Korean ASCII EUC	X'AFFE'
970	971	Korean ASCII EUC	X'AFFE'
1381	1380	S-Chinese ASCII GB-Code	X'FEFE'
1383	1382	S-Chinese ASCII EUC	X'A1A1'
1386	1385	S-Chinese ASCII EUC	X'A1A1'
5026	4396	Japanese EBCDIC	X'FEFE'
5035	4396	Japanese EBCDIC	X'FEFE'
5039	1351	Japanese ASCII	X'FFFD'

The equivalence of SBCS and DBCS characters depends on M.

Each character of the argument determines a character of the result. Regardless of the character set identified by M, 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 *n*th character of the argument is a DBCS character, the *n*th character of the result is that DBCS character.
- If the *n*th character of the argument is an SBCS character that has an equivalent DBCS character, the *n*th character of the result is that equivalent DBCS character.

- If the *n*th character of the argument is an SBCS character that does not have an equivalent DBCS character, the *n*th character of the result is the DBCS substitution character.
- In the ASCII environment, if the last byte of the argument is a DBCS introducer byte, the last 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 16336.

If the second argument is not specified,

G G
- If the expression is an empty string, the length attribute of the result is 1. In DB2 UDB for UWO the length attribute is 0.
- Otherwise, the length attribute of the result is the same as the length attribute of the first argument.

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.

The CCSID of the result is the CCSID of *graphic-expression*.

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 109.

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.

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 (‘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
  2000-12-31    54
  ```

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see "String Representations of Datetime Values" on page 49.

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
  2000-12-31    52
  ```

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 with an actual length that is not greater than 255 bytes. For the valid formats of string representations of dates and timestamps, see “String Representations of Datetime Values” on page 49.
- 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 100.

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
  ```

Chapter 4. Queries

A *query* specifies a result table or an intermediate result table. A query is a component of certain SQL statements. The three forms of a query are the *subselect*, the *fullselect*, and the *select-statement*. There is another SQL statement that can be used to retrieve at most a single row described under "SELECT INTO" on page 457.

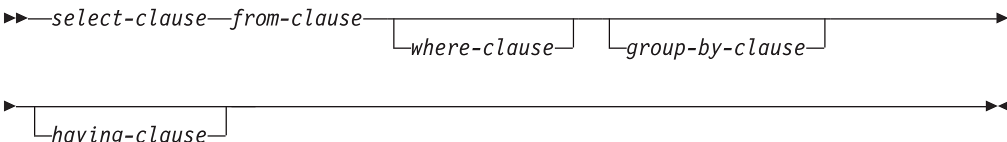
Authorization

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, one of the following:
  - The SELECT privilege on the table or view
  - Ownership of the table or view
- Administrative authority.

© Copyright IBM Corp. 1982, 2002

subselect

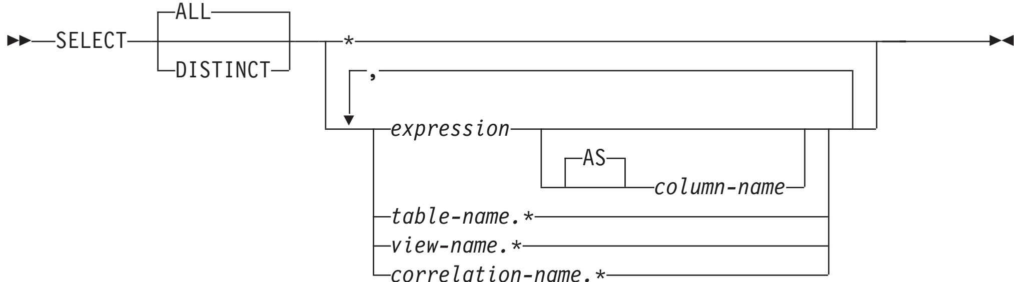


The *subselect* is a component of the fullselect and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a subselect.

A 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 logical sequence of the 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
Retains all rows of the final result table and does not eliminate redundant duplicates. This is the default.

DISTINCT
Eliminates all but one of each set of duplicate rows of the final result table. Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value in the second row. (For determining duplicate rows, two null values are considered equal.)

DISTINCT must not be used more than once in a subselect. This restriction includes SELECT DISTINCT and the use of DISTINCT in a column function of the select list or HAVING clause, but does not include subqueries of the subselect.

When SELECT DISTINCT is specified, no column in the list of column names can be a VARCHAR with length greater than 255, a VARGRAPHIC with length greater than 127 or a LOB.

Select list notation

*****
Represents a list of 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. Therefore, * 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 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 an exposed table, view, or correlation name in the FROM clause immediately following the SELECT clause. The first name in the

list identifies the first column of the table or view, the second name in the list identifies the second column of the table or view, and so on.

The list of names is established when the statement containing the SELECT clause is prepared. Therefore, * does not identify any columns that have been added to a table after the statement has been prepared.

In all the products, SQL statements can be implicitly or explicitly rebound (prepared again). The effect of a rebind on statements that include * or name.* is as follows:

G G G
- In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO, the list of names is reestablished. Therefore, the number of columns returned by the statement may change.

G G G G
- In DB2 UDB for iSeries, the list of names is normally not reestablished. Therefore, the number of columns returned by the statement will not change. There are cases, however, where the list of names is reestablished. See the product documentation for details.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared). The number of columns in the list must not exceed 750. See Table 41 on page 511 for more information.

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 either identify a grouping column or be specified within a column function.
- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the column functions in the select list.

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, it is applied to each row of R, and the result contains as many rows as there are rows in R.
- If the select list is a list of column functions, R is the source of the arguments of the functions and the result of applying the select list is one row.

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 or COUNT_BIG
- A column that allows null values
- A scalar function or expression with an operand that allows nulls
- A host variable that has an indicator variable, or in the case of Java, a host variable or expression whose type is able to represent a Java null value

- A result of a UNION if at least one of the corresponding items in the select list is nullable
- An arithmetic expression in an outer select list
- A user-defined scalar 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 the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), then the result column name is the unqualified name of that column.
- All other result column names 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.
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	see "Expressions" on page 96 for a description of data type attributes.
any function	The data type of the result of the function. For a built-in function, see Chapter 3, "Built-in Functions" on page 129 to determine the data type of the result. For a user-defined function, the data type of the result is what was defined in the CREATE FUNCTION statement for the function.
the name of any string column	the same as the data type of the column, with the same length attribute.
the name of any string host variable	the same as the data type of the variable, with a length attribute equal to the length of the variable. If the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
a character-string constant of length *n*	VARCHAR(*n*)
a graphic-string constant of length *n*	VARGRAPHIC(*n*)
the name of a datetime column	the same as the data type of the column.

When the expression is:	The data type of the result column is:
the name of a distinct type column	the same as the distinct type of the column, with the same length, precision, and scale attributes, if any.

from-clause

The FROM clause specifies an intermediate result table.

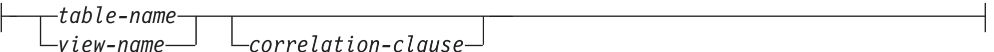
If only one *table-reference* is specified, the intermediate result table is simply the result of that *table-reference*. If more than one *table-reference* is specified in the FROM clause, the intermediate result table consists of all possible combinations of the rows of the specified *table-reference* (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

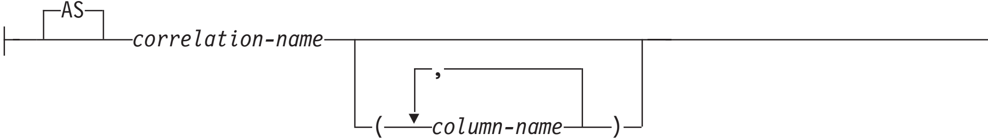
single-table:

table-name / view-name [correlation-clause]

nested-table-expression:



correlation-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 subselect in parenthesis is called a *nested table expression*.[47] 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 *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 241.

47. A *nested-table-expression* is also called a *derived table*.

Each *table-name* and *view-name* must identify an existing table or view at the current server.

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*.

The exposed names of all table references must 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, or nested table expression must use the exposed name. If the same table name or view name is specified twice, at least one specification must be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table, view or *nested-table-expression*. When a *correlation-name* is specified, column-names can also be specified to give names to the columns of the *table-name*, *view-name*, or *nested-table-expression*. 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*. For more information, see "Correlation Names" on page 79.

In general, *nested-table-expressions* can be specified in any FROM clause. Columns from the nested table expressions can be referenced in the select list and in the rest of the subselect using the correlation name which must be specified. The scope of this correlation name is the same as correlation names for other table or view names in the FROM clause. A nested table expression can be used:
- in place of a view to avoid creating the view (when general use of the view is not required)
- when the desired result table is based on host variables.

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 81

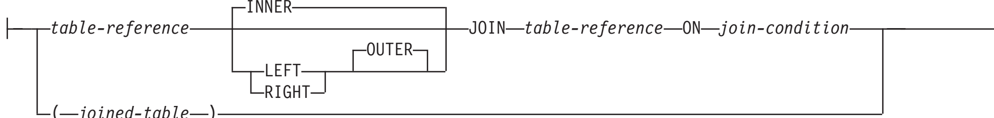
The following example is valid:

```
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                  INCORRECT
  FROM DEPARTMENT D,
  (SELECT AVG(E.SALARY) AS  AVGSAL, COUNT (*) AS EMPCOUNT
    FROM EMPLOYEE E
    WHERE E.WORKDEPT = D.DEPTNO ) AS EMPINFO
```

joined-table



A *joined-table* specifies an intermediate result table that is the result of either an inner join or outer join. The table is derived by applying one of the join operators: INNER, LEFT OUTER or RIGHT OUTER to its operands.

If a join-operator is not specified, INNER is implicit. The order in which multiple joins are specified 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 each row of the right table keeping only the rows where the join-condition is true. Thus, the result table may be missing rows from either or both of the joined tables. Outer joins include the rows produced by the inner join as well as the missing rows, depending on the type of outer join as follows:

- A left outer join includes the rows from the left table that were missing from the inner join.
- A right outer join includes 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 any subqueries.
- One *expression* in each predicate can either use a column from the table specified to the right of the JOIN keyword, or can contain no columns.
- The other *expression* can use columns from any of the tables specified in the current *from-clause* prior to the JOIN keyword, or can contain no columns.
- 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*.
- It cannot include an SQL function.

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 79 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 of an operand table. 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 join 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.

where-clause

```
►►──WHERE──search-condition──────────────────────────────────────────►◄
```

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the subselect.

The *search-condition* must conform to the following rules:

- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, 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 references is typically executed just once.

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.

Each *column-name* must unambiguously identify a column of R other than a column that is a VARCHAR with length greater than 255, a VARGRAPHIC with length greater than 127 or a LOB. Each identified column is called a *grouping column*.

The result of GROUP BY is a set of groups of rows. In each group of more than one row, all values of each grouping column are equal; and all rows with the same set of values of the grouping columns are in the same group. For grouping, all null values within a grouping column are considered equal.

Because every row of a group contains the same value of any grouping column, the name of a grouping column can be used in a search condition in a HAVING clause or an expression in a SELECT clause. In each case, the reference specifies only one value for each group.

If the grouping column contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and may not all have the same length. In that case, a reference to the grouping column still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

GROUP BY cannot be used in a subquery of a basic predicate or if R is derived from a view whose outer subselect includes GROUP BY or HAVING clauses.

The number of columns must not exceed 120 and the sum of their length attributes must not exceed 2000. See Table 41 on page 511 for more information.

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 columns.

Each *column-name* in the search condition must do one of the following:

- Unambiguously identify a grouping column of R.
- Be specified within a column function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a table, view, or *nested-table-expression* identified in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each column function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see "Example 6" on page 246 and "Example 7" on page 246.

A correlated reference to a group of R must either identify a grouping column or be contained within a column function.

The HAVING clause must not be used in a subquery of a basic predicate or if R is derived from a view whose outer subselect includes GROUP BY or HAVING clauses. When HAVING is used without GROUP BY, any column name in the select list must appear within a column function.

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
```

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.

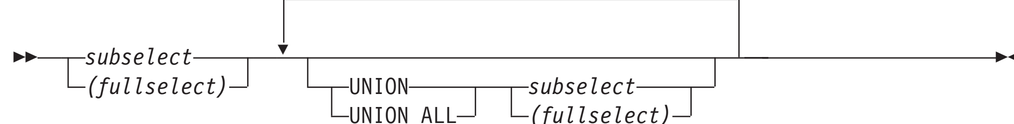
Example 8

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 INSERT 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 or **UNION ALL**
: Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the redundant 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 R 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.)

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 parenthesis 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. If UNION is specified without the ALL option, R1 and R2 must not include a column that is a VARCHAR with length greater than 255, a VARGRAPHIC with length greater than 127 or a LOB.

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 68.

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 EMPPROJACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

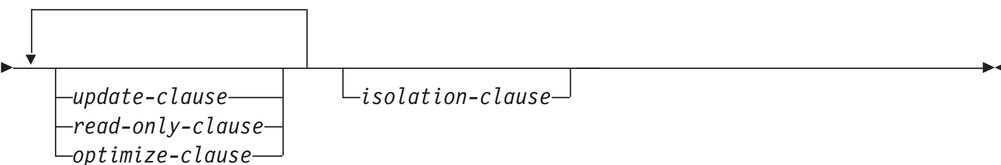
```
SELECT EMPNO FROM EMPLOYEE
  WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO FROM EMPPROJACT
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

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 EMPPROJACT
  WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

select-statement

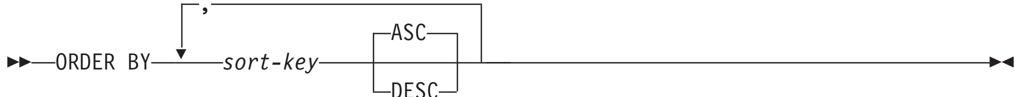


Notes:

1. The update-clause and read-only-clause cannot both be specified in the same select-statement.
2. The update-clause and order-by-clause cannot both be specified in the same select-statement.
3. 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 of any of the database managers. In any case, the result 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 sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on. A column that is a VARCHAR with length greater than 255, a VARGRAPHIC with length greater than 127 or a LOB must not be identified.

A named column in the select list may be identified by a *sort-key* that is an *integer* or a *column-name*. An unnamed column in the select list must be identified by an *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 237 defines when result columns are unnamed. If the fullselect includes a UNION operator, see "fullselect" on page 248 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 the ordering specification does not determine a complete ordering, rows with duplicate key values 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-key*s must not exceed 1012 and the sum of their length attributes must not exceed 4000. See Table 41 on page 511 for more information.

column-name
: Must unambiguously identify a column of the result table. The column must not be a LOB column. Although columns not included in the result table must not be referenced in the ORDER BY clause, 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 81 for more information.

  If the fullselect includes a UNION or UNION ALL, the column-name must not 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. An error is returned if the subselect:

  - specifies DISTINCT in the select-clause
  - includes column functions in the select list
  - includes a GROUP BY clause

integer
Must be greater than 0 and not greater than the number of columns in the result table. The integer *n* identifies the *n*th column of the result table. This column must not be a LOB 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* specified in these clauses must exactly match an expression in the select list, except that blanks are not significant.

A *sort-key-expression* may not reference column names specified in an AS clause in the SELECT list.

The *sort-key-expression* cannot include a non-deterministic function or a function with an external action. The *sort-key-expression* must not be a LOB.

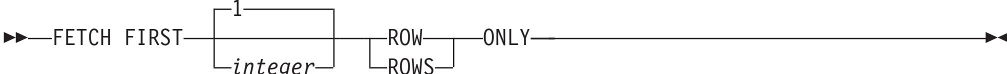
A *sort-key-expression* cannot be specified if DISTINCT is used in the select list of the subselect.

If the subselect is grouped, the *sort-key-expression* can be an expression in the select list of the subselect or can include a column function, constant or host variable.

ASC
Uses the values of the column in ascending order. This is the default.

DESC
Uses the values of the column in descending order.

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 the order-by-clause and the fetch-first-clause are both 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

```
>>--FOR UPDATE--+--------------------------+-------------------------><
                |        .-,-----------.   |
                |        V             |   |
                '--OF----column-name---+---'
```

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 a dynamically prepared *select-statement* does not contain an UPDATE clause, the cursor associated with that *select-statement* cannot be referenced in a Positioned UPDATE statement.

If a statically prepared *select-statement* does not contain an UPDATE clause and its result table is not read-only, Positioned UPDATE statements identifying the cursor associated with that *select-statement* can update all updatable columns. [48]

G If an UPDATE clause is not specified in a statically prepared *select-statement* used
G by an DB2 UDB for iSeries or DB2 UDB for UWO application requester connected
G to a DB2 UDB for z/OS and OS/390 application server, its associated cursor
G cannot be referenced in a Positioned UPDATE statement.

48. In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, a program preparation option must be used if the UPDATE clause is not specified and the cursor is referenced in subsequent Positioned UPDATE statements. If this program preparation option is not used and the UPDATE clause is not specified, the cursor cannot be referenced in a Positioned UPDATE statement. For DB2 UDB for z/OS and OS/390 the program preparation option is STDSQL(YES) or NOFOR, for DB2 UDB for UWO it is LANGLEVEL SQL92E.

read-only-clause

```
►►──FOR READ ONLY──────────────────────────────────────────────────►◄
```

The READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements.

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 READ ONLY or ORDER BY clause, the database manager might open cursors as if the UPDATE clause was specified.

A read-only result table must not be referred to in an UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY.

Syntax alternatives: FOR FETCH ONLY can be specified in place of FOR READ ONLY.

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, the database manager assumes that all rows of the result table will 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 optional isolation-clause specifies the isolation level at which the select statement is executed.

- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

WITH UR can be specified only if the result table is read-only. If *isolation-clause* is not specified, the default isolation is used with the exception of a default isolation level of uncommitted read. With uncommitted read, the default isolation level of the statement depends on whether the result table is read-only; if the result table is read-only then the default will be UR; if the result table is not read-only then the default will be CS. See “Isolation Level” on page 13 for a description of how the default is determined.

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) AS AVGSAL
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY AVGSAL
```

Example 4

Declare a cursor named UP_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT PROJNO, PRSTDATE, PRENDATE
              FROM PROJECT
              FOR UPDATE OF PRSTDATE, PRENDATE;
```

Example 5

Select items from a table with an isolation level of Repeatable Read (RS).

```
SELECT NAME, SALARY
  FROM PAYROLL
  WHERE DEPT = 704
  WITH RS
```

Chapter 5. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements listed in the following tables.

Table 25. SQL Schema Statements

SQL Statement	Description	Reference
ALTER TABLE	Alters the description of a table	267
COMMENT	Adds or replaces a comment to the description of an object	289
CREATE ALIAS	Creates an alias	303
CREATE DISTINCT TYPE	Creates a distinct type	304
CREATE FUNCTION	Creates a user-defined function (introduction)	310
CREATE FUNCTION (External Scalar)	Creates an external scalar function	314
CREATE FUNCTION (Sourced)	Creates a user-defined function based on another existing scalar or column function	325
CREATE FUNCTION (SQL Scalar)	Creates an SQL scalar function	332
CREATE INDEX	Creates an index on a table	338
CREATE PROCEDURE	Creates a procedure (introduction)	340
CREATE PROCEDURE (External)	Creates an external procedure	341
CREATE PROCEDURE (SQL)	Creates an SQL procedure	348
CREATE TABLE	Creates a table	353
CREATE TRIGGER	Creates a trigger	368
CREATE VIEW	Creates a view of one or more tables or views	376
DROP	Drops an object	395
GRANT (Distinct Type Privileges)	Grants privileges on a distinct type	410
GRANT (Function or Procedure Privileges)	Grants privileges on a function or procedure	412
GRANT (Package Privileges)	Grants privileges on a package	416
GRANT (Table or View Privileges)	Grants privileges on a table or view	418
RENAME	Renames a table	442
REVOKE (Distinct Type Privileges)	Revokes the privilege to use a distinct type	444
REVOKE (Function or Procedure Privileges)	Revokes privileges on a function or procedure	446
REVOKE (Package Privileges)	Revokes the privilege to execute statements in a package	450

© Copyright IBM Corp. 1982, 2002

Table 25. SQL Schema Statements (continued)

SQL Statement	Description	Reference
REVOKE (Table or View Privileges)	Revokes privileges on a table or view	452

Table 26. SQL Data Change Statements

SQL Statement	Description	Reference
DELETE	Deletes one or more rows from a table	386
INSERT	Inserts one or more rows into a table	423
UPDATE	Updates the values of one or more columns in one or more rows of a table	466

Table 27. SQL Data Statements

SQL Statement	Description	Reference
	All SQL Data Change statements	Table 26
CLOSE	Closes a cursor	287
DECLARE CURSOR	Defines an SQL cursor	381
FETCH	Positions a cursor on a row of the result table and assigns values from the row to host variables	406
FREE LOCATOR	Removes the association between a LOB locator variable and its value	409
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table	428
OPEN	Opens a cursor	429
SELECT	Executes a query	456
SELECT INTO	Assigns values to host variables	457
SET transition-variable	Assigns values to transition variables in a trigger	464
VALUES	Provides a way to invoke a user-defined function from a trigger	472
VALUES INTO	Specifies a result table of no more than one row and assigns the values to host variables	473

Table 28. SQL Transaction Statements

SQL Statement	Description	Reference
COMMIT	Ends a unit of work and commits the database changes made by that unit of work	294
ROLLBACK	Ends a unit of work and backs out the database changes made by that unit of work	455

Table 29. SQL Connection Statements

SQL Statement	Description	Reference
CONNECT (Type 1)	Connects to a server and establishes the rules for remote unit of work	296

Table 29. SQL Connection Statements (continued)

SQL Statement	Description	Reference
CONNECT (Type 2)	Connects to a server and establishes the rules for application-directed distributed unit of work	300
RELEASE (Connection)	Places one or more connections in the release-pending state	440
SET CONNECTION	Establishes the server of the process by identifying one of its existing connections	459

Table 30. SQL Dynamic Statements

SQL Statement	Description	Reference
DESCRIBE	Describes the result columns of a prepared statement	391
EXECUTE	Executes a prepared SQL statement	401
EXECUTE IMMEDIATE	Prepares and executes an SQL statement	404
PREPARE	Prepares an SQL statement for execution	433

Table 31. SQL Session Statements

SQL Statement	Description	Reference
SET PATH	Assigns a value to the CURRENT PATH special register	461

Table 32. SQL Embedded Host Language Statements

SQL Statement	Description	Reference
BEGIN DECLARE SECTION	Marks the beginning of an SQL declare section	281
CALL	Calls a procedure	283
END DECLARE SECTION	Marks the end of an SQL declare section	400
INCLUDE	Inserts declarations into a source program	421
WHENEVER	Defines actions to be taken on the basis of SQL return codes	475

Table 33. SQL Control Statements

SQL Statement	Description	Reference
assignment-statement	Assigns a value to an output parameter or to a local variable	480
CALL	Calls a procedure	481
CASE	Selects an execution path based on multiple conditions	482
compound-statement	Groups other statements together in an SQL routine	484
GET DIAGNOSTICS	Obtains information about the previous SQL statement that was executed	491
GOTO	Branches to a user-defined label within an SQL routine or trigger	493

Table 33. SQL Control Statements (continued)

SQL Statement	Description	Reference
IF	Provides conditional execution based on the truth value of a condition	494
LEAVE	Continues execution by leaving a block or loop	496
LOOP	Repeats the execution of a statement or group of statements	497
REPEAT	Executes a statement or group of statements until a search condition is true	498
RESIGNAL	Resignals an error or warning condition	500
RETURN	Returns from a routine	502
SIGNAL	Signals an error or warning condition	504
WHILE	Repeats the execution of a statement while a specified condition is true	506

How SQL Statements Are Invoked

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The *Invocation* section in the description of each statement indicates whether or not the statement is executable.

An *executable statement* can be invoked in three ways:
- Embedded in an application program
- Dynamically prepared and executed
- Issued interactively.

Note: Statements embedded in REXX are prepared and executed dynamically.

Depending on the statement, some or all of these methods can be used. The *Invocation* section in the description of each statement tells which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

Embedding a Statement in an Application Program

SQL statements can be included in a source program that will be submitted to the precompiler. 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 and COBOL, each embedded statement must be preceded by the keywords EXEC and SQL. For more information, see Appendix H, “Coding SQL Statements in C Applications” on page 593 and Appendix I, “Coding SQL Statements in COBOL Applications” on page 609.
- In Java, each embedded statement must be preceded by the keywords #sql. For more information, see Appendix J, “Coding SQL Statements in Java Applications” on page 627.
- In REXX, each embedded statement must be preceded by the keyword EXECSQL. For more information, see Appendix K, “Coding SQL Statements in REXX Applications” on page 643.

Executable statements

An executable statement embedded in an application program is executed every time a statement of the host language would be executed if specified in the same place. Thus, a statement within a loop is executed every time the loop is executed, and a statement within a conditional construct is executed only when the condition is satisfied.

An embedded statement can contain references to host variables. A host variable referenced in this way can be used in two ways:

- As input (the current value of the host variable is used in the execution of the statement)
- As output (the variable is assigned a new value as a result of executing the statement).

In particular, all references to host variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input. The treatment of other references is described individually for each statement.

All executable statements should be followed by a test of an SQL return code. Alternatively, the WHENEVER statement (which is itself nonexecutable) can be used to change the flow of control immediately after the execution of an embedded statement.

Objects referenced in SQL statements need not exist when the statements are bound (statically prepared). [49]

Nonexecutable statements

An embedded nonexecutable statement is processed only by the precompiler. The precompiler reports any errors encountered in the statement. The statement is *never* executed, and acts as a no-operation if placed among executable statements of the application program. Therefore, such statements should not be followed by a test of an SQL return code.

Dynamic Preparation and Execution

An application program can dynamically build an SQL statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the program (for example, input from a workstation). In C, COBOL, and REXX, the statement can be prepared for execution by means of the (embedded) statement PREPARE and executed by means of the (embedded) statement EXECUTE. Alternatively, the (embedded) statement EXECUTE IMMEDIATE can be used to prepare and execute a statement in one step. 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. Instead, the statement can contain parameter markers. See “PREPARE” on page 433 for rules concerning the parameter markers. When the prepared statement is executed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. See “EXECUTE” on page 401 for rules concerning this replacement. Once prepared, a statement can

49. In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, a program preparation option is available to allow reference to objects that do not exist when the SQL statements are bound.

be executed several times with different values of host variables. Parameter markers are not allowed in EXECUTE IMMEDIATE.

In C, COBOL, and REXX, the successful or unsuccessful execution of the statement is indicated by the setting of an SQL return code after the EXECUTE (or EXECUTE IMMEDIATE) statement. The SQL return code should be checked as described above. See "SQL Return Codes" for more information. In Java, the successful or unsuccessful execution of the statement is handled by Java Exceptions. For more information see "Handling SQL Errors and Warnings in Java" on page 639.

Static Invocation of a select-statement

A *select-statement* can be included as a part of the (nonexecutable) statement DECLARE CURSOR. Such a statement is executed every time the cursor is opened by means of the (embedded) statement OPEN. After the cursor is open, the result table can be retrieved one row at a time by successive executions of the FETCH statement.

Used in this way, the *select-statement* can contain references to host variables. These references are effectively replaced by the values that the variables have at the moment of executing OPEN.

Dynamic Invocation of a select-statement

An application program can dynamically build a *select-statement* in the form of a character string placed in a host variable. In general, the statement is built from some data available to the program (for example, a query obtained from a workstation). The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and referenced by a (nonexecutable) statement DECLARE CURSOR. The statement is then executed every time the cursor is opened by means of the (embedded) statement OPEN. After the cursor is open, the result table can be retrieved one row at a time by successive executions of the FETCH statement.

Used in this way, the *select-statement* must not contain references to host variables. It can contain parameter markers instead. See "PREPARE" on page 433 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 429 for rules concerning this replacement.

Interactive Invocation

A capability for entering SQL statements from a workstation is part of the architecture of the database manager. A statement entered in this way is said to be issued interactively.

A statement issued interactively must be an executable statement that does not contain parameter markers or references to host variables, because these make sense only in the context of an application program.

SQL Return Codes

Each host language provides a mechanism for handling SQL return codes:

- In C or COBOL, 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.

– A stand-alone integer variable named SQLCODE.

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 and an integer variable named SQLCODE .

A stand-alone SQLSTATE should be used to conform with the SQL 1999 Core standard.[50]

- 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. For more information, see "Handling SQL Errors and Warnings in Java" on page 639.
- In REXX, an SQLCA is provided automatically.

SQLSTATE

The SQLSTATE is 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 format of the SQLSTATE values is the same for all database managers and is consistent with the SQL 1999 Core standard. See Appendix E, "SQLSTATE Values—Common Return Codes" on page 539 for more information and a complete list of the possible values of SQLSTATE.

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.

SQLCODE values may provide additional product-specific information about an error or warning. Portable applications should use SQLSTATE values instead of SQLCODE values.

50. In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO, a program preparation option must be used to indicate the use of a stand-alone SQLCODE. In DB2 UDB for z/OS and OS/390, the same option must be used to indicate the use of a stand-alone SQLSTATE. For DB2 UDB for z/OS and OS/390 use the precompiler option STDSQL(YES). For DB2 UDB for UWO, use the program preparation option LANGLEVEL SQL92E.

SQL Comments

In C and COBOL, static SQL statements can include host language or SQL comments. In Java and REXX, static SQL statements cannot include host language or SQL comments. For more information, see Appendix J, “Coding SQL Statements in Java Applications” on page 627 and Appendix K, “Coding SQL Statements in REXX Applications” on page 643.

SQL comments are introduced by two hyphens.

These rules apply to the use of SQL comments:[51]

- The two hyphens must be on the same line, not separated by a space.
- Comments can be started wherever a space is valid (except within a delimiter token or between “EXEC” and “SQL”).
- Comments cannot be continued on the next line.
- Comments are not allowed within statements that are dynamically prepared (using PREPARE or EXECUTE IMMEDIATE).
- In COBOL, the hyphens must be preceded by a space.

Example: This example shows how to include comments in 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
```

51. In DB2 UDB for z/OS and OS/390, the precompiler option STDSQL(YES) must be used to allow SQL comments.

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:
- The ALTER privilege for the table
- Administrative authority.

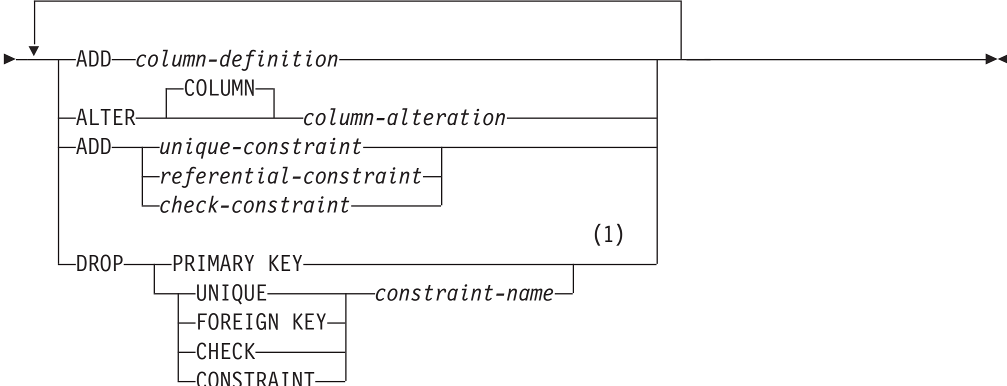
To define a foreign key, the privileges held by the authorization ID of the statement must include at least one of the following on the parent table:
- The REFERENCES privilege on the table
- The REFERENCES privilege on each column of the specified parent key
- Administrative authority.

To drop the primary key of table T, the privileges held by the authorization ID of the statement must include at least one of the following on every table that is a dependent of T:
- The ALTER privilege on the table
- Administrative authority

To refer to a distinct type, the privileges held by the authorization ID of the statement must include at least one of the following:
- USAGE privilege on the distinct type
- Administrative authority

Syntax

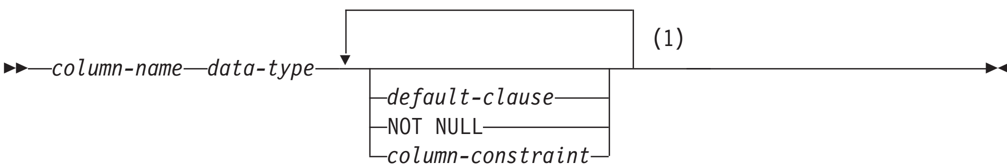
```
►►──ALTER TABLE──table-name──────────────────────────────────────────────►
```



Notes:

1 The same clause must not be specified more than once, except for the ALTER COLUMN clause, which can be specified more than once. Do not specify DROP CONSTRAINT if DROP FOREIGN KEY or DROP CHECK is specified.

column-definition:



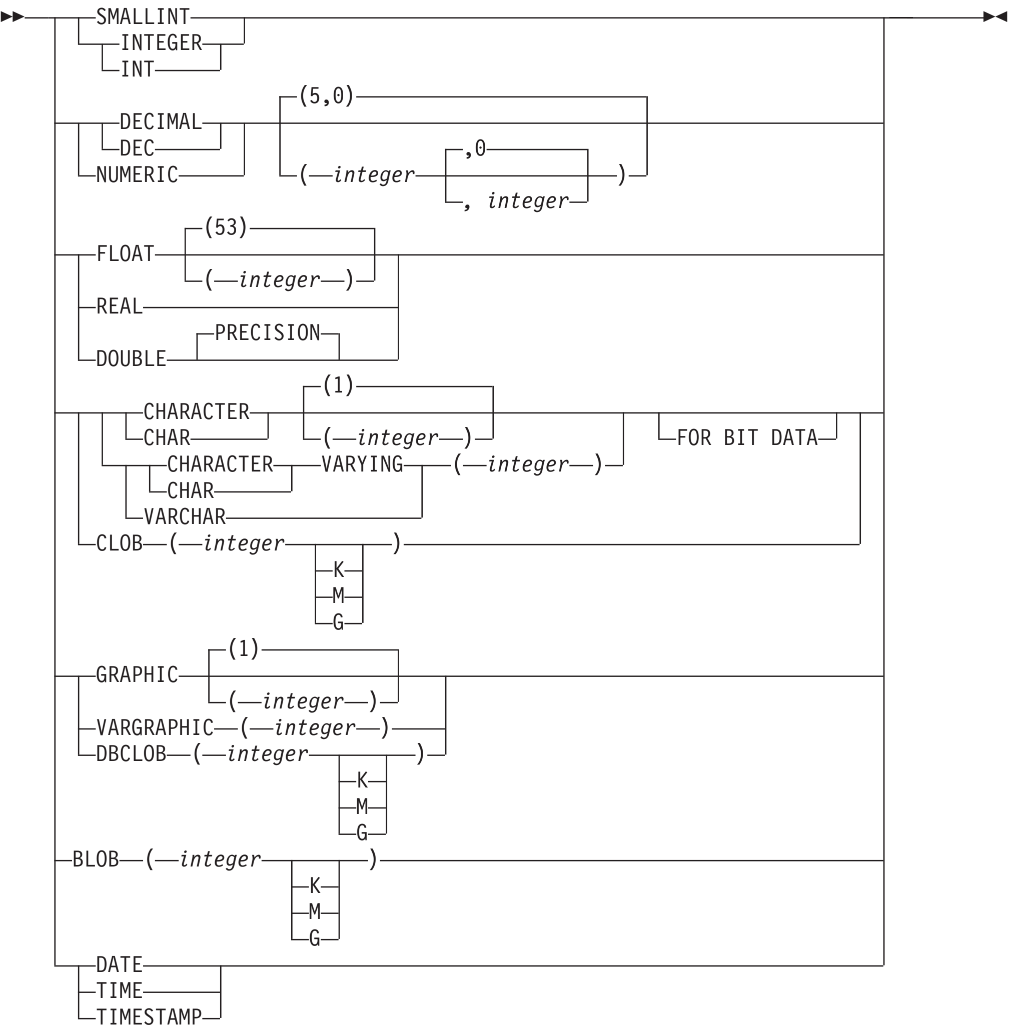
Notes:

1 The same clause must not be specified more than once.

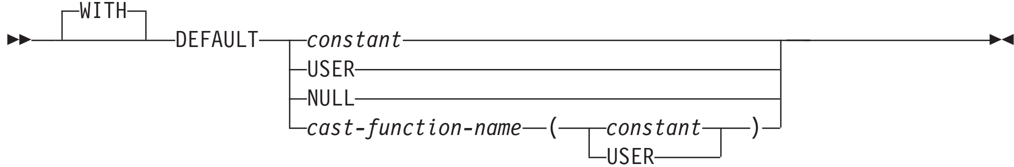
data-type:

```
>>--+-built-in-type------+---------------------------------------><
    '-distinct-type-name-'
```

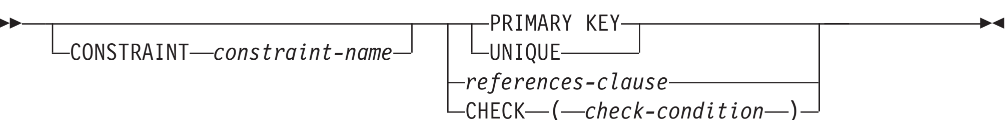
built-in-type:



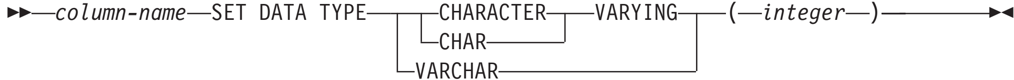
default-clause:



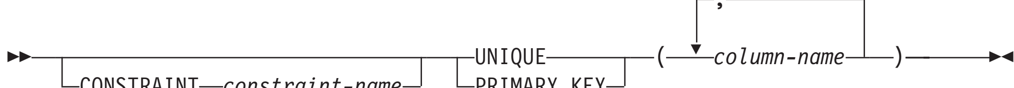
column-constraint:



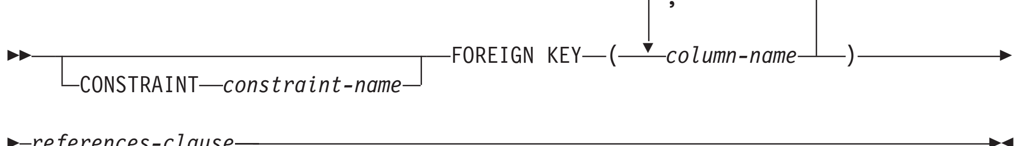
column-alteration:



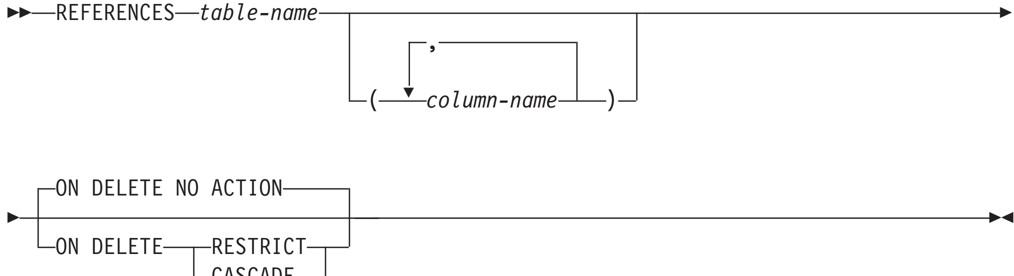
unique-constraint:



referential-constraint:



references-clause:



check-constraint:

Description

table-name
: Identifies the table to be altered. The *table-name* must identify a table that exists at the current server. It must not be a view or a catalog 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. If the table previously had *n* columns, the ordinality of the new column is *n*+1. The value of *n*+1 must not exceed 750. [52] See Table 41 on page 511 for more information.

Adding the new column must not make the total byte count of all columns exceed the maximum record size. The maximum record size is 32 677. See Table 41 on page 511 for more information.

G For DB2 UDB for z/OS and OS/390, to add a LOB column the table must already
G have a ROWID column. For more information, see the product documentation.

column-definition

column-name
Names the column to be added to the table. Do not use the same name for more than one column name of the table. Do not qualify *column-name*.

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 353 for the 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 in the SQL path.

DEFAULT
Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

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 58. 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 as the default for the column. The data type of the column or 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 the

52. This value is 1 less if the table is a dependent table.

USER special register. For existing rows, the value is that of the USER special register at the time the ALTER TABLE statement is processed.

NULL
Specifies null as the default for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same *column-definition*.

cast-function-name
Specifies the name of the cast function that matches the name of the distinct type name of the data type for the column.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type. This form of the DEFAULT value can only be used with columns that are defined as a 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.

USER
Specifies the value of the USER special register at the time of INSERT as the default for the column. 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 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.

If the value specified is not valid, an error is returned.

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 be the same as a constraint name that was previously specified in the ALTER 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
G specified as a separate clause. PRIMARY KEY is not supported by DB2
G UDB for z/OS and OS/390 for *column-constraint*.

The NOT NULL clause must be specified with this 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 column.

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. UNIQUE is not supported by DB2 UDB for z/OS and OS/390 for *column-constraint*.

The NOT NULL clause must be specified with this clause. This clause cannot be specified more than once in a column definition and must not be specified if the PRIMARY KEY clause is specified in the column definition. The column must not be a LOB 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. The column must not be a LOB column. For more information, see "REFERENCES clause" on page 275.

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. For more information, see "CHECK clause" on page 277.

End of column-definition

ALTER column-alteration

Alters the definition of a column. A column cannot be altered if it is used in a view or referential constraint.

column-alteration

column-name
Identifies the column to be altered. The name must not be qualified and must identify an existing column in the table that has a VARCHAR data type. The name must not identify a column that is being added in the same ALTER TABLE statement.

SET DATA TYPE CHARACTER VARYING *(integer)* or **SET DATA TYPE CHAR VARYING** *(integer)* or **SET DATA TYPE VARCHAR** *(integer)*

Increases the length of an existing VARCHAR column. The data type of *column-name* must be VARCHAR and the current maximum length defined for the column must not be greater than the value for *integer*. The value for *integer* may range up to the maximum length for a VARCHAR, 32 672. See Table 41 on page 511 for more information.

Altering the column must not make the total byte count of all columns exceed the maximum record size. The maximum record size is 32 677. See Table 41 on page 511 for more information.

If the column is used in a unique constraint or an index, the new length must not cause the sum of the stored lengths for the unique constraint or index to exceed 255. See Table 41 on page 511 for more information.

End of column-alteration

ADD unique-constraint

unique-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.

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 column. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. See Table 41 on page 511 for more information.

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. The identified columns must be defined as NOT NULL. Any existing values in the set of columns must be unique.

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.

G In DB2 UDB for z/OS and OS/390, the unique index must already exist.

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 number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. See Table 41 on page 511 for more information. The table must not already have a primary key.

The identified columns cannot be the same as the set of 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. The identified columns must be defined as NOT NULL. Any existing values in the set of columns must be unique.

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.

G In DB2 UDB for z/OS and OS/390, the unique index must already exist.

End of unique-constraint

ADD referential-constraint

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 column. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. See Table 41 on page 511 for more information.

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.

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
G constraints are allowed, but not recommended. In DB2 UDB for z/OS and
G OS/390, duplicate referential constraints are ignored with a warning.

Let T2 denote the identified parent table.

G In DB2 UDB for z/OS and OS/390, if T1 and T2 are the same table, ON
G DELETE CASCADE or ON DELETE NO ACTION must be specified.

(*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 column. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 255. See Table 41 on page 511 for more information.

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 table must have a unique index with a key that is identical to the primary key. The keys are identical only if they have the same number of columns and the *n*th column name of one is the same as the *n*th column name of the other. 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 other attributes.

If a foreign key column is a distinct type, the data type of the corresponding column of the parent key must have the same distinct type.

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
G execution of the ALTER TABLE statement. In DB2 UDB for z/OS and OS/390,
G the table space of a non-empty table is placed in a check pending status. Therefore, every 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 four possible actions:

- NO ACTION (default) [53]
- RESTRICT
- CASCADE
- SET NULL

SET NULL must not be specified unless some column of the foreign key allows null values. SET NULL must not be specified if T1 has an update trigger.

CASCADE must not be specified if T1 has a delete trigger.

G In DB2 UDB for UWO and DB2 UDB for z/OS and OS/390, a
G self-referencing table with a SET NULL or RESTRICT rule must not be a
G dependent in a referential constraint with a delete rule of CASCADE.

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 is returned and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the relationship would form a cycle and T2 is already

53. In DB2 UDB for z/OS and OS/390, the default depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the default is RESTRICT. If the value is 'SQL', the default is NO ACTION.

delete-connected to T1, then the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table, or
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2, or
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3, or
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3,

If *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

End of referential-constraint

ADD check-constraint

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.[54]

The *check-condition* is a form of the *search-condition*, except:

- It can only refer to columns of the table whose data type is not a LOB data type or a distinct type based on a LOB data type.
- It can be up to 3800 bytes long, not including redundant blanks. See Table 41 on page 511 for more information.
- It must not contain any of the following:
  - subqueries
  - built-in functions
  - column functions
  - host variables

54. In DB2 UDB for z/OS and OS/390, the value of the CURRENT RULES special register must be 'STD' to get this behavior.

- parameter markers
- special registers
- user-defined functions (except cast functions generated for distinct types)
- CASE expressions

G G In DB2 UDB for z/OS and OS/390, the *check-condition* is subject to additional restrictions. See the product reference for further information.

For more information about *search-condition*, see “Search Conditions” on page 126.

End of check-constraint

DROP

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.

If a primary index was implicitly created to support uniqueness of the primary key, it is dropped.

DROP FOREIGN KEY *constraint-name*
Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint in which the table is a dependent.

DROP 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.

If a primary index was implicitly created to support uniqueness of the primary key, it is dropped.

DROP CHECK *constraint-name*

Drops the check constraint *constraint-name*. The *constraint-name* must identify a check constraint on the table.

DROP CONSTRAINT *constraint-name*
Drops the constraint *constraint-name*. The constraint-name must identify a primary key, unique, referential, or check constraint in 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.

DROP CONSTRAINT must not be used in the same ALTER TABLE statement as DROP PRIMARY KEY, DROP UNIQUE KEY, DROP FOREIGN KEY or DROP CHECK.

Notes

Columns not automatically added to views: Any columns added via ALTER TABLE will not automatically be added to any existing view of the table.

G G **Invalidation of access plans:** Adding or dropping primary, foreign or unique keys or check constraints or altering column lengths may invalidate access plans. The rules are product-specific.

G G G **Names of indexes created automatically:** The rules for generating the name of an index that is created during the execution of the ALTER TABLE statement are product-specific.

Order of operations: The order of operations within an ALTER TABLE statement is product-specific.

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  BLOB(1K) 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
  ADD CONSTRAINT DEPTQUIP
    FOREIGN KEY (EQUIP_OWNER)
      REFERENCES DEPARTMENT
      ON DELETE SET NULL
```

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 $16,000.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT REVENUE
  CHECK (SALARY + COMM > 16000)
```

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. An SQL declare section contains declarations of host variables that are eligible to be used as host variables in SQL statements in a program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX.

Authorization

None required.

Syntax

```
►►──BEGIN DECLARE SECTION──────────────────────────────────────────────────►◄
```

Description

The BEGIN DECLARE SECTION statement 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 400.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and must not be nested.

SQL statements must not be included within an SQL declare section, with the exception of INCLUDE statements that include host variable declarations.

Host variables referenced in SQL statements must be declared in an SQL declare section in all host languages, other than Java and REXX. Furthermore, the declaration of each variable must appear before the first reference to the variable. Host variables are declared without the use of these statements in Java, and they are not declared at all in REXX.

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.

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];
         }                            hv_vchar24;
  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

The CALL statement calls a procedure.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

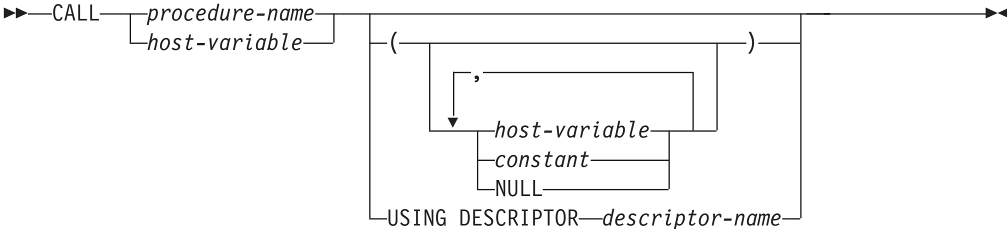
Authorization

The authorization ID of the statement must have at least one of the following:

- The EXECUTE privilege on the procedure
- Ownership of the procedure
- Administrative authority

G In DB2 UDB for UWO, these privileges must be held by the run-time authorization
G ID on the package associated with the procedure.

Syntax



Description

procedure-name **or** *host-variable*
: Identifies the procedure to call by the specified *procedure-name* or the procedure name contained in the *host-variable*. The identified procedure must exist at the current server.

  If a *host-variable* is specified:

  - It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 254 bytes.
  - It must not be followed by an indicator variable.
  - The value within the host variable must be left justified and must not contain any embedded blanks.
  - If the host variable is a fixed length string, the value within the host variable must be padded on the right with blanks if its length is less than that of the host variable.
  - The value within the host variable must be in uppercase characters unless the procedure name 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 38.

The procedure definition at the current server determines the name of the external program, language, and calling convention of the procedure. See "CREATE PROCEDURE" on page 340 for more information.

host-variable or *constant* or **NULL**

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 CREATE PROCEDURE) 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 73 and "References to Host Variables" on page 85. NULL specifies the null value.

G When the application is running on a DB2 UDB for UWO application
G requestor, all parameters must be host variables.

USING DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of host variables that are passed as parameters to the procedure. If the procedure has no parameters, the SQLDA is ignored.

Before the CALL statement is processed, the user must set the following fields in the SQLDA (Note that the rules for REXX are different. For more information, see Appendix K, "Coding SQL Statements in REXX Applications" on page 643):

- 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 on 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 529.

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 for the procedure. The *n*th variable described by the SQLDA corresponds to the *n*th parameter in the procedure.

The USING DESCRIPTOR clause is not supported for a CALL statement within a Java program.

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 58.

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.[55]

Result Sets from procedures: Any cursors specified using the WITH RETURN clause that the procedure leaves open when it returns identifies a result set. In a procedure written in Java, all cursors are implicitly defined WITH RETURN.

Results sets are returned only when the procedure is called from CLI, JDBC, or SQLJ. If the procedure was invoked from CLI or Java, and more than one cursor is left open, the result sets can only be processed in the order in which the cursors were opened. Only unread rows are available to be fetched. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the procedure at the time the procedure is terminated, then rows 151 through 500 will be returned to the stored procedure.

Locks in procedures: All locks that have been acquired in the called procedure are retained until the end of the unit of work.

Errors from Procedures: A procedure can return errors (or warnings) using the SQLSTATE like other SQL statements. Applications should be aware of the possible SQLSTATEs that can be expected when invoking a procedure. The possible SQLSTATEs depend on how the procedure is coded. Procedures may also return SQLSTATEs such as those that begin with '38' or '39' if the database manager encounters problems executing the procedure. Applications should therefore be prepared to handle any error SQLSTATE that may result from issuing a CALL statement.

Nesting CALL statements: A program that is executing as a procedure can issue a CALL statement. When a procedure calls another procedure, the call is considered to be nested. If a nested procedure returns a result set, the result set is available only to the immediate caller of the nested procedure.

Examples

Example 1: Call procedure PGM1 and pass two parameters.

```
CALL PGM1 (:hv1,:hv2)
```

Example 2: In C, invoke a procedure called SALARY_PROCED using the SQLDA named INOUT_SQLDA.

55. Product-specific options exist that may extend the scope of cursors and prepared statements.

```
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 381 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: At the end of a unit of work, all open cursors declared without the WITH HOLD option that belong to an application process are implicitly closed.

Close cursors for performance: Explicitly closing cursors as soon as possible can improve performance.

Procedure considerations: Special rules apply to cursors within procedures that have not been closed before returning to the calling program. For more information, see "CALL" on page 283.

Examples

In a COBOL program, use the cursor C1 to fetch the values from the first four columns of the EMPPROJACT 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 EMPPROJACT                        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 a comment in the catalog descriptions of an object.

Invocation

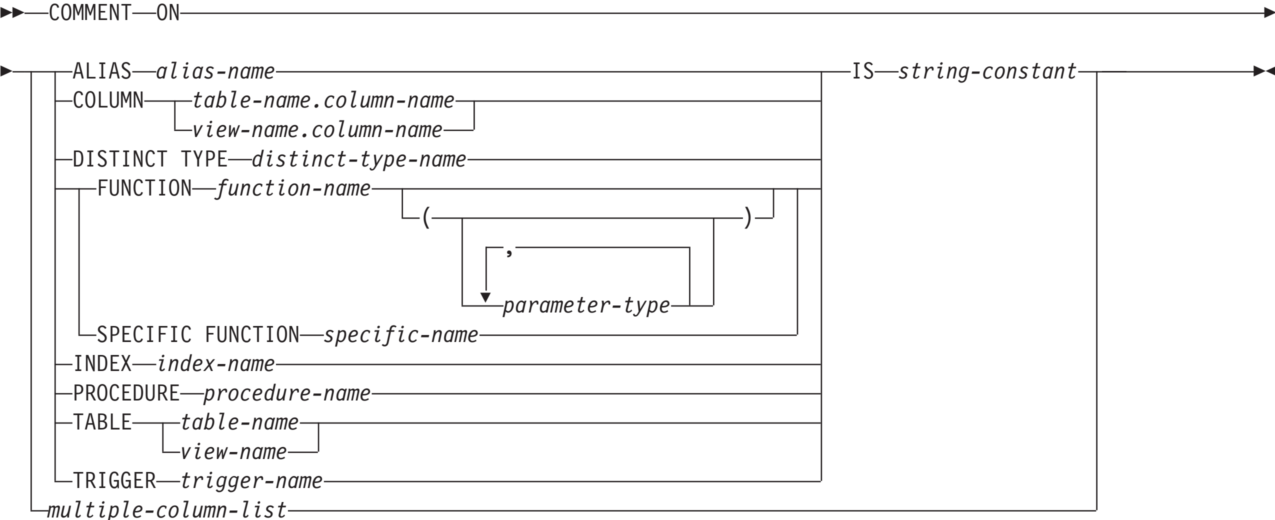
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the object
- Administrative authority.

Syntax



multiple-column-list:

```
              ┌─,──────────────────────────────┐
├──┬─table-name─┬──(───▼─column-name──IS──string-constant─┴──)──────────────┤
   └─view-name──┘
```

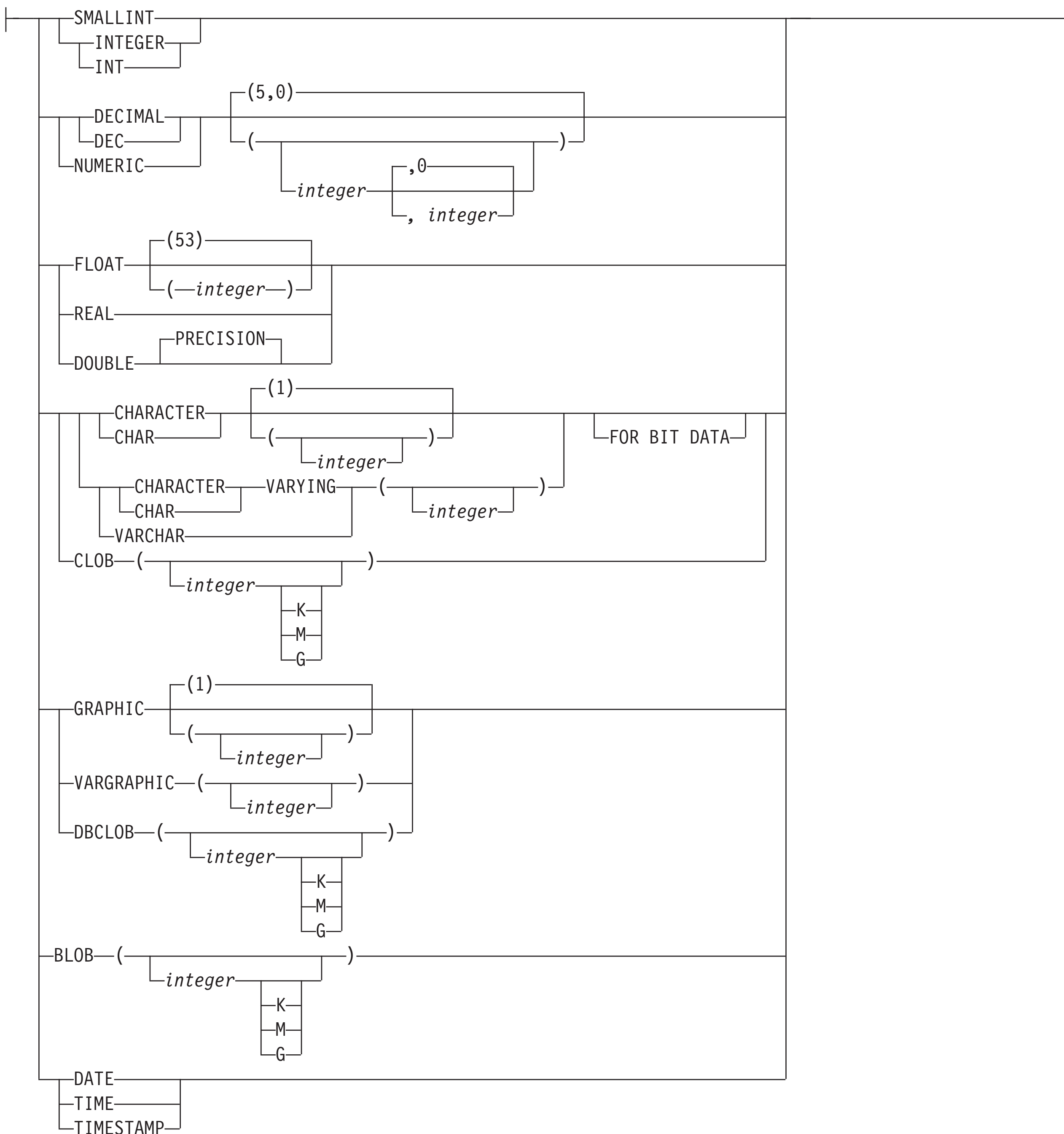
parameter-type:

```
├──data-type──┬────────────┬──────────────────────────────────────────────┤
              └─AS LOCATOR─┘
```

data-type:

```
├──┬─built-in-type──────┬─────────────────────────────────────────────────┤
   └─distinct-type-name─┘
```

built-in-type:



Description

ALIAS *alias-name*
: Identifies the alias to which the comment applies. *alias-name* must identify an alias that exists at the current server.

COLUMN
: 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, and 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. *distinct-type-name* must identify a distinct type that exists at the current server.

FUNCTION or SPECIFIC FUNCTION
Identifies the function to which the comment applies. The function must exist at the current server and it must be a function that was defined with the CREATE FUNCTION statement or a cast function that was generated by a CREATE DISTINCT TYPE statement. The particular 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 to which the comment applies. Synonyms for data types are considered a match. The rules for function resolution (and the SQL path) are not used.

If *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 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). 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.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

INDEX *index-name*
Identifies the index to which the comment applies. *index-name* must identify an index that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure to which the comment applies. *procedure-name* must identify a procedure that exists at the current server.

TABLE *table-name* or *view-name*
Identifies the table or view to which the comment applies. The name must identify a table or view that exists at the current server.

TRIGGER *trigger-name*
Identifies the trigger to which the comment applies. *trigger-name* must identify a trigger that exists at the current server.

IS
Introduces the comment to be added or replaced.

string-constant
Can be any character string constant of up to 254 characters.

multiple-column-list
To comment on more than one column in a table or view with a single COMMENT statement, specify the table or view name, followed by a list in parenthesis of the form:

```
(column-name IS string-constant,
 column-name IS string-constant, ... )
```

Each column name must not be qualified, and must identify a column of the specified table or view that exists at the current server.

Examples

Example 1: Add a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
  IS 'Reflects first quarter 2000 reorganization'
```

Example 2: Add a comment for the EMP_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1
  IS 'View of the EMPLOYEE table without salary information'
```

Example 3: Add a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
  IS 'highest grade level passed in school'
```

Example 4: Add comments for two different columns of the DEPARTMENT table.

```
COMMENT ON DEPARTMENT
  (MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
   ADMRDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT')
```

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.

Authorization

None required.

Syntax

Description

The COMMIT statement ends the unit of work in which it is executed. It commits all changes made by SQL schema statements and SQL data change statements during the unit of work. For more information see Chapter 5, "Statements" on page 259.

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.

Effect of commit: Commit causes the following to occur:

- Connections in the release-pending state are ended. Some products provide options that cause remote connections in the held state to be 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.
- For DB2 UDB for z/OS and OS/390, prepared statements are destroyed, except those statements required for the cursors that were not closed.

Other transaction environments: SQL COMMIT may not be available in other transaction environments, such as IMS and CICS. To do a commit operation in these environments, SQL programs must use the call prescribed by their transaction manager.

Examples

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;
      return;
    SQLERR:
      ...
      EXEC SQL WHENEVER SQLERROR CONTINUE; /* continue if error on rollback */
      EXEC SQL ROLLBACK;
      return;
  }
```

CONNECT (Type 1)

The CONNECT (Type 1) statement connects an application process to the identified application server and establishes the rules for remote unit of work. This server is then the current server for the process. Differences between this statement and the CONNECT (Type 2) statement are described in Appendix G, “CONNECT (Type 1) and CONNECT (Type 2) Differences” on page 591. Refer to “Application-Directed Distributed Unit of Work” on page 20 for more information about connection states.

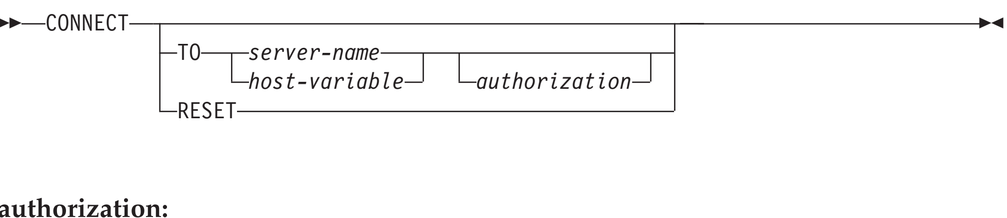
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

The authorization ID of the statement must be authorized to connect to the identified application server. The authorization check is performed by that server.
G The authorization required is product-specific.

Syntax



authorization:

```
>>--USER--host-variable--USING--host-variable--------------------><
```

Description

TO *server-name* or *host-variable*
Identifies the application 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 CHAR or VARCHAR variable with a length attribute that is not
G greater than 18. In DB2 UDB for z/OS and OS/390, the maximum length is
G 16. In DB2 UDB for UWO, the maximum length is 8.
- 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.
- 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 an application server described in the local directory and the application process must be in the

G G G connectable state. (See "Notes" for information about connection states.) In DB2 UDB for UWO, the server name is a database alias name identifying the application server.

USER *host-variable*
Identifies the authorization name that will be used to connect to the application server. The *host-variable* must satisfy the following:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 8. See Table 41 on page 511 for more information.
- 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 for 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 authorization name must not contain lowercase characters.

G G For DB2 UDB for z/OS and OS/390, authorization may not be specified when the connection type is IMS or CICS.

USING *host-variable*
Identifies the password that will be used to connect to the application server. The *host-variable* must satisfy the following:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 8. See Table 41 on page 511 for more information.
- 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 the length 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.

G G G For DB2 UDB for UWO, CONNECT RESET only disconnects the application process from the current server. If implicit connect is available, the application process remains unconnected until an SQL statement is issued.

CONNECT with no operand
This form of the CONNECT statement returns information about the current server and has no effect on connection states. The information is returned in the SQLERRP field of the SQLCA as described below.

Notes

Successful Connection: If the CONNECT statement (excluding the CONNECT with no operand form) is successful:

- All open cursors are closed, all prepared statements are destroyed, all locators are freed, and all locks are released from the previous application server.
- The application process is disconnected from its previous application server, if any, and connected to the identified application server.
- The name of the application server is placed in the CURRENT SERVER special register.
- Information about the application server is placed in the SQLERRP field of the SQLCA. The format below applies if the application server is a DB2 Universal Database product. The information has the form *pppvvrrm*, where:

- *ppp* is:
  - DSN for DB2 UDB for z/OS and OS/390
  - QSQ for DB2 UDB for iSeries
  - SQL for DB2 UDB for UWO
- *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 z/OS and OS/390, the value would be 'DSN07010'.

- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.

G

Unsuccessful Connection: 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 identifies the product. For example, if the application requester is DB2 UDB for UWO, the first three characters are 'SQL'.

If the CONNECT statement is unsuccessful because the application process is not in the connectable state, the connection state of the application process is unchanged.

G G G

If the CONNECT statement is unsuccessful because the *server-name* is not listed in the local directory, the connection state of the application process is product-specific.

If the CONNECT statement is unsuccessful for any other reason, the application process is placed in the unconnected state, all open cursors are closed, all prepared statements are destroyed, and any held resources are released.

For a description of connection states, see "Remote Unit of Work Connection Management" on page 18. See the description of the CONNECT statement in your product's SQL reference for further information.

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 (strncmp(SQLSTATE, "00000", 5) )
    { strncpy(product,sqlca.sqlerrp,3); }
  ...
  return;
}
```

CONNECT (Type 2)

The CONNECT (Type 2) statement connects the application process to the identified application server and establishes the rules for application-directed distributed unit of work. This server is then the current server for the process. Differences between this statement and the CONNECT (Type 1) statement are described in Appendix G, "CONNECT (Type 1) and CONNECT (Type 2) Differences" on page 591. Refer to "Application-Directed Distributed Unit of Work" on page 20 for more information about connection states.

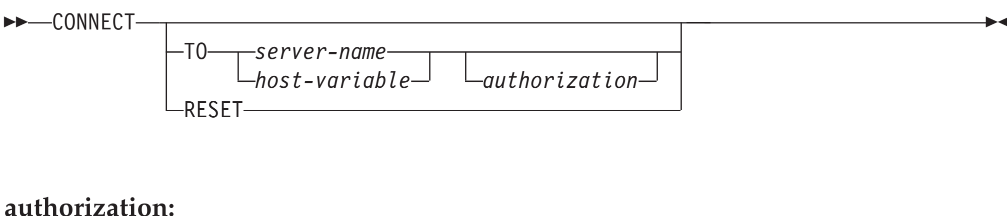
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

The authorization ID of the statement must be authorized to connect to the identified application server. The authorization check is performed by that server. The authorization required is product-specific.

Syntax



authorization:

```
►►──USER──host-variable──USING──host-variable──────────────────────►◄
```

Description

TO *server-name* or *host-variable*

Identifies the application 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 CHAR or VARCHAR variable with a length attribute that is not greater than 18. In DB2 UDB for z/OS and OS/390, the maximum length is 16. In DB2 UDB for UWO, the maximum length is 8.
- 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.
- 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 an application server described in the local directory.

Let S denote the specified server name or the server name contained in the host variable. The application process must not have an existing connection to S.[56]

USER *host-variable*
Identifies the authorization name that will be used to connect to the application server. The *host-variable* must satisfy the following:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 8. See Table 41 on page 511 for more information.
- 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 for 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 authorization name must not contain lowercase characters.

USING *host-variable*
Identifies the password that will be used to connect to the application server. If the *host-variable* is specified:

- It must be a CHAR or VARCHAR variable with a length attribute that is not greater than 8. See Table 41 on page 511 for more information.
- 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 the length of the host variable, it must be padded on the right with blanks.
- The value of the password must not contain lowercase characters.

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. The information is returned in the SQLERRP field of the SQLCA as described below.

Notes

Successful Connection: If the CONNECT statement (excluding the CONNECT with no operand form) is successful:

- A connection to application server S is created and placed in the current and held states. The previously current connection, if any, is placed in the dormant state.
- S is placed in the CURRENT SERVER special register.
- Information about application server S is placed in the SQLERRP field of the SQLCA. The format below applies if the application server is a DB2 Universal Database product. The information has the form *pppvvrrm*, where:
  - *ppp* is:

    DSN for DB2 UDB for z/OS and OS/390

    QSQ for DB2 UDB for iSeries

    SQL for DB2 UDB for UWO

56. In DB2 UDB for z/OS and OS/390, this rule is enforced only if the SQLRULES(STD) bind option is specified.

– *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 z/OS and OS/390, the value would be 'DSN07010'.

- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.

G

Unsuccessful Connection: If the CONNECT statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.

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)
```

CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table or view.

Invocation

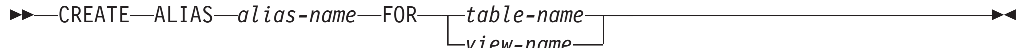
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority.

Syntax



Description

alias-name
: Names the alias. The name, including the implicit or explicit qualifier, must not be the same as an index, table, view or alias that already exists at the current server.

  If the *alias-name* is qualified, the schema name must not be a system schema.

FOR *table-name* or *view-name*
: Identifies the table or view at the current server for which *alias-name* is defined. An alias name must not be specified (an alias cannot refer to another alias).

  An alias can be defined for an object that does not exist at the time of the definition. If it does not exist when the alias is created, a warning is returned. However, the referenced object must exist when a SQL statement containing the alias is used, otherwise an error is returned.

Examples

Example: Create an alias named CURRENT_PROJECTS for the PROJECT table.

```
CREATE ALIAS CURRENT_PROJECTS
  FOR PROJECT
```

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

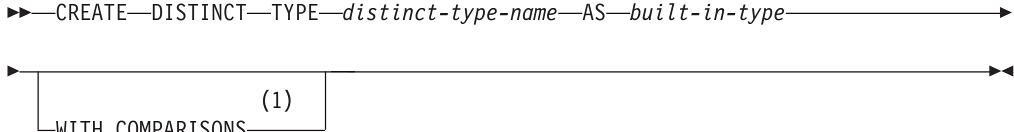
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

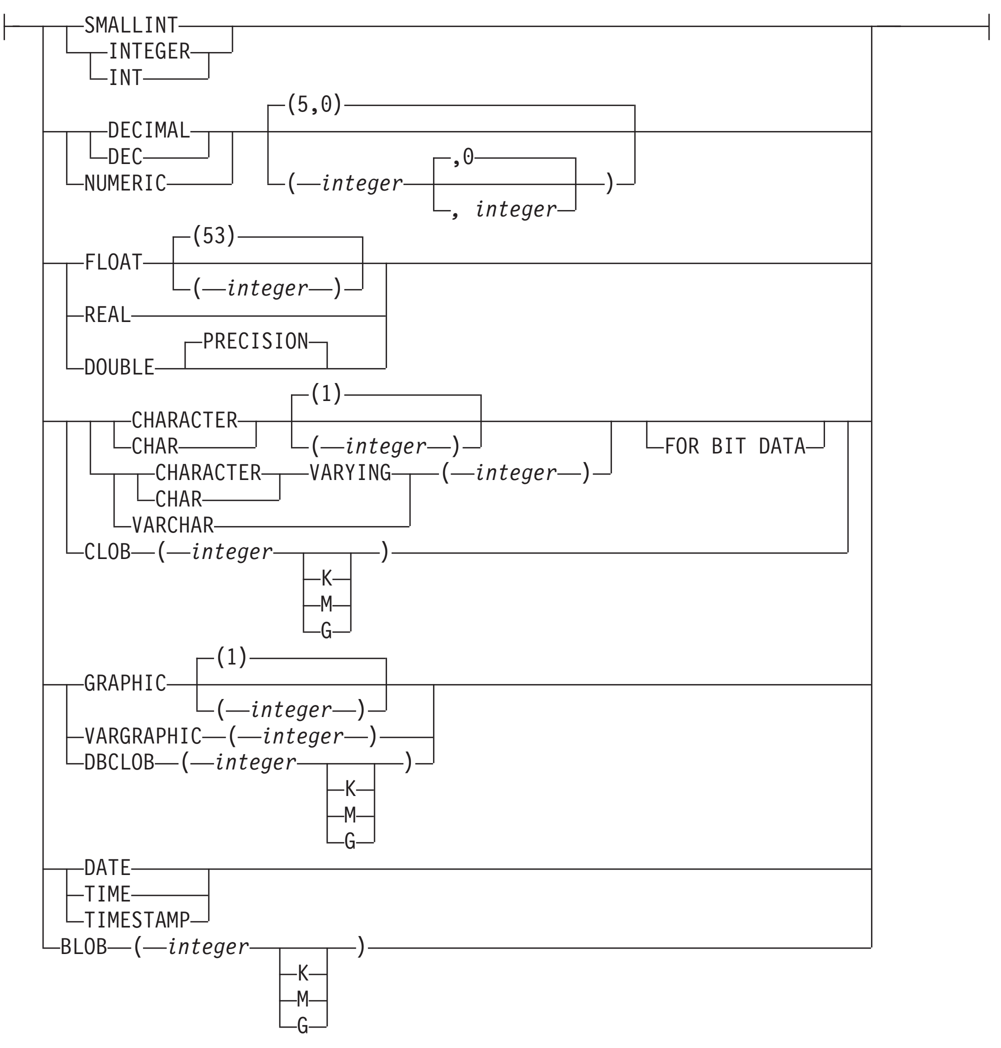
The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority

Syntax



built-in-type:



Notes:

1 Specify WITH COMPARISONS for built-in data types except for BLOB, CLOB, and DBCLOB.

Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier must not identify a distinct type that already exists at the current server. *distinct-type-name* must not be the same as the name of a built-in data type, or any of the following, even they are specified as delimited identifiers:

```
ALL        NODENAME    TRUE
AND        NODENUMBER  TYPE
ANY        NOT         UNIQUE
BETWEEN    NULL        UNKNOWN
BOOLEAN    ONLY        WHEN
CASE       OR          =
CAST       OVERLAPS    ¬=
CHECK      PARTITION   <
DISTINCT   POSITION    <=
EXCEPT     RRN         ¬<
EXISTS     SELECT      >
FALSE      SIMILAR     >=
FOR        SOME        ¬>
```

```
FROM         STRIP         !<
IN           SUBSTRING     <>
IS           TABLE         !>
LIKE         THEN          !=
MATCH        TRIM
```

If a qualified *distinct-type-name* is specified, the schema name must not be one of the system schemas (see “Schemas” on page 3).

built-in-type

Specifies the built-in data type used as the basis for the internal representation of the distinct type. See “CREATE TABLE” on page 353 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 a specific value is not specified 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.

WITH COMPARISONS

Specifies that system-generated comparison operators are to be created for comparing two instances of a distinct type. These keywords should not be specified if the built-in type is BLOB, CLOB, or DBCLOB, otherwise a warning will be returned and the comparison operators will not be generated. For all other *built-in-type*s, the WITH COMPARISONS keywords are required. When a distinct type is created using the WITH COMPARISONS clause, the database manager allows the comparison operators with the exception of LIKE and NOT LIKE. In order to use the LIKE predicate on a distinct type, it must be cast to a built-in type. The comparison operators are invoked as infix operators, not by using functional notation; that is, C1 < C2, not ″<″(C1,C2).

Notes

Owner Privileges: The owner of the distinct type is authorized to define columns, parameters, or variables with the distinct type with the ability to grant these privileges to others. See “GRANT (Distinct Type Privileges)” on page 410. The owner is also authorized to invoke the generated cast functions (see “GRANT (Function or Procedure Privileges)” on page 412). For more information on ownership of the object, see “Authorization, Privileges and Object Ownership” on page 27.

Additional Generated Functions: Besides the system-generated comparison operators described above, the following functions become available to convert to, and from the source type:

- The distinct type to the source type
- The source type to the distinct type
- INTEGER to the distinct type if the source type is SMALLINT
- DOUBLE to distinct type if the source type is REAL
- VARCHAR to the distinct type if the source type is CHAR
- VARGRAPHIC to the distinct type if the source type is GRAPHIC

These functions are created as if the following statements were executed:

```
CREATE FUNCTION source-type-name (distinct-type-name)
   RETURNS source-type-name ...

CREATE FUNCTION distinct-type-name (source-type-name)
   RETURNS distinct-type-name ...
```

Names of the Generated Cast Functions: Table 34 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 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 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 including the schema qualfier. 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 as the generated cast function must not already exist in the current server.

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.

The following table gives the names of the functions to convert from the distinct type to the source type and from the source type to the distinct type for all predefined data types.

Table 34. CAST functions on distinct types

Source Type Name	Function Name	Parameter-type	Return-type
SMALLINT	*distinct-type-name*	SMALLINT	*distinct-type-name*
	distinct-type-name	INTEGER	*distinct-type-name*
	SMALLINT	*distinct-type-name*	SMALLINT
INTEGER	*distinct-type-name*	INTEGER	*distinct-type-name*
	INTEGER	*distinct-type-name*	INTEGER
DECIMAL	*distinct-type-name*	DECIMAL (*p*,*s*)	*distinct-type-name*
	DECIMAL	*distinct-type-name*	DECIMAL (*p*,*s*)
G NUMERIC or DECIMAL[1]	*distinct-type-name*	NUMERIC (*p*,*s*) or DECIMAL (*p*,*s*)	*distinct-type-name*
G	NUMERIC or DECIMAL	*distinct-type-name*	NUMERIC (*p*,*s*) or DECIMAL (*p*,*s*)
REAL or FLOAT(*n*) where *n* defines a single precision floating point number	*distinct-type-name*	REAL	*distinct-type-name*
	distinct-type-name	DOUBLE	*distinct-type-name*
	REAL	*distinct-type-name*	REAL

Table 34. CAST functions on distinct types (continued)

Source Type Name	Function Name	Parameter-type	Return-type
DOUBLE or DOUBLE PRECISION or FLOAT or FLOAT(*n*) where *n* defines a double precision floating point number	*distinct-type-name*	DOUBLE	*distinct-type-name*
	DOUBLE	*distinct-type-name*	DOUBLE
CHAR	*distinct-type-name*	CHAR (*n*)	*distinct-type-name*
	CHAR	*distinct-type-name*	CHAR (*n*)
	distinct-type-name	VARCHAR (*n*)	*distinct-type-name*
VARCHAR	*distinct-type-name*	VARCHAR (*n*)	*distinct-type-name*
	VARCHAR	*distinct-type-name*	VARCHAR (*n*)
CLOB	*distinct-type-name*	CLOB (*n*)	*distinct-type-name*
	CLOB	*distinct-type-name*	CLOB (*n*)
GRAPHIC	*distinct-type-name*	GRAPHIC (*n*)	*distinct-type-name*
	GRAPHIC	*distinct-type-name*	GRAPHIC (*n*)
	distinct-type-name	VARGRAPHIC (*n*)	*distinct-type-name*
VARGRAPHIC	*distinct-type-name*	VARGRAPHIC (*n*)	*distinct-type-name*
	VARGRAPHIC	*distinct-type-name*	VARGRAPHIC (*n*)
DBCLOB	*distinct-type-name*	DBCLOB (*n*)	*distinct-type-name*
	DBCLOB	*distinct-type-name*	DBCLOB (*n*)
BLOB	*distinct-type-name*	BLOB (*n*)	*distinct-type-name*
	BLOB	*distinct-type-name*	BLOB (*n*)
DATE	*distinct-type-name*	DATE	*distinct-type-name*
	DATE	*distinct-type-name*	DATE
TIME	*distinct-type-name*	TIME	*distinct-type-name*
	TIME	*distinct-type-name*	TIME
TIMESTAMP	*distinct-type-name*	TIMESTAMP	*distinct-type-name*
	TIMESTAMP	*distinct-type-name*	TIMESTAMP

Note:

G 1. When the source data type is specified as NUMERIC, whether a separate function named NUMERIC is generated
G is platform-specific. DB2 UDB for iSeries generates a cast function named NUMERIC. DB2 UDB for z/OS and
G OS/390 and DB2 UDB for UWO do not generate a cast function named NUMERIC, use DECIMAL instead.

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 built-in function, has been created for the distinct type. See “Extending or Overriding a Built-in Function” on page 312.

The schema name of the distinct type must be included in the SQL path 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 314.
- **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 value, or the result of a column function. See "CREATE FUNCTION (Sourced)" on page 325. 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 332.

Notes

Choosing the Schema and Function Name: If a qualified function name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 3). If *function-name* is not qualified, it is implicitly qualified with the default schema name.

The unqualified function name must not be the same as the name of a built-in data type, or any of the following, even they are specified as delimited identifiers:

```
ALL           NODENAME       TRUE
AND           NODENUMBER     TYPE
ANY           NOT            UNIQUE
BETWEEN       NULL           UNKNOWN
BOOLEAN       ONLY           WHEN
CASE          OR             =
CAST          OVERLAPS       ¬=
CHECK         PARTITION      <
DISTINCT      POSITION       <=
EXCEPT        RRN            ¬<
EXISTS        SELECT         >
FALSE         SIMILAR        >=
FOR           SOME           ¬>
FROM          STRIP          !<
IN            SUBSTRING      <>
IS            TABLE          !>
LIKE          THEN           !=
MATCH         TRIM
```

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. The input and result parameters specified and the implicit parameters for indicators, SQLSTATE, function name, specific name, and message text are included. See Appendix A, "SQL Limits" on page 509 for more details on the limits. DB2 UDB for z/OS and OS/390 only uses the first 30 parameters to determine uniqueness.

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 "Rules for Result Data Types" on page 68. For example, a constant that is one of the input arguments to the function might have a built-in data type that is different from the data type that the function expects, and more significantly, might not be promotable to that expected data type. Based on the rules of promotion, using the following data types is recommended:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC

  For portability of functions across platforms, use the following recommended data type names:
  - DOUBLE or REAL instead of FLOAT.
  - DECIMAL instead of NUMERIC.
- **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 and only when LANGUAGE JAVA is not in effect.

  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 must not be specified for a sourced or SQL function.

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, or scale) 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. If the function has more than 30 parameters, DB2 UDB for z/OS and OS/390 only considers the first 30 parameters to determine whether the function is unique.

G
G
G

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 external 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. See "CREATE FUNCTION (Sourced)" on page 325 for more information. 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, it might be useful to use different rules for rounding than the built-in ROUND function. 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.

  Once a built-in function has been overridden, if the schema for the new function appears in the SQL path before the system schemas, the data base manager may

choose a user-defined function rather than the built-in function. 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. This can occur with dynamic SQL statements, or when static SQL applications are rebound. For more information on when static statements are rebound, see “Packages and Access Plans” on page 9.

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)

The 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

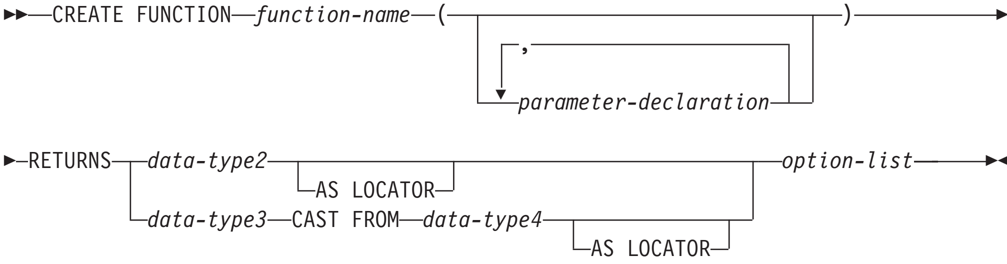
The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority.

For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Administrative authority.

Syntax

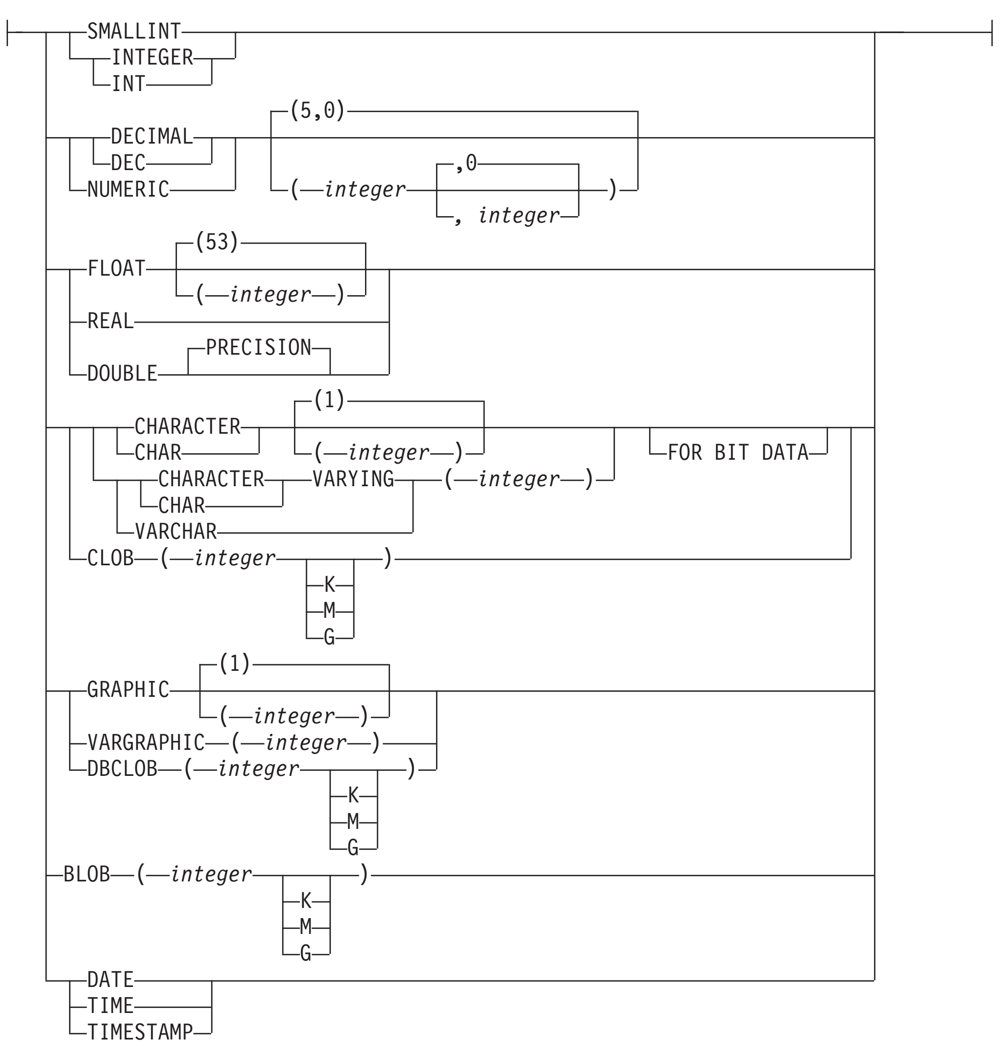


parameter-declaration:

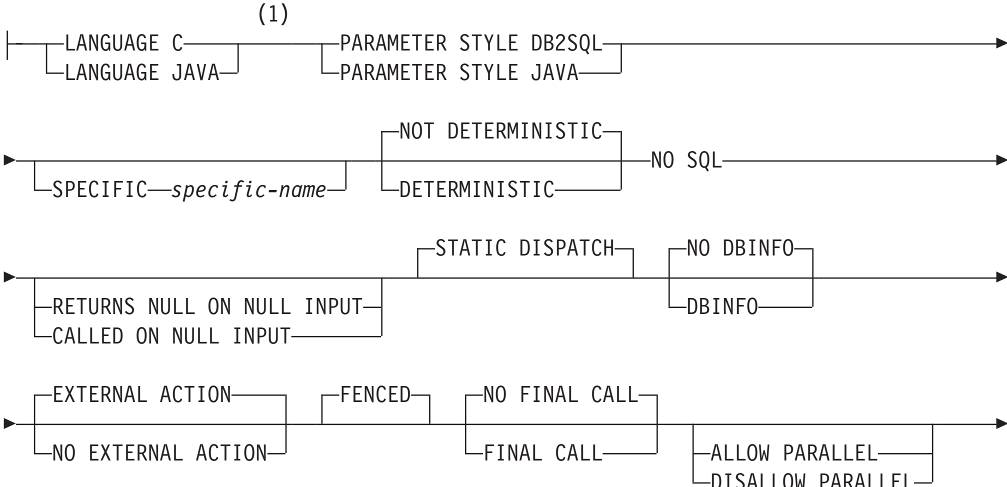
data-type1, data-type2, data-type3, data-type4:

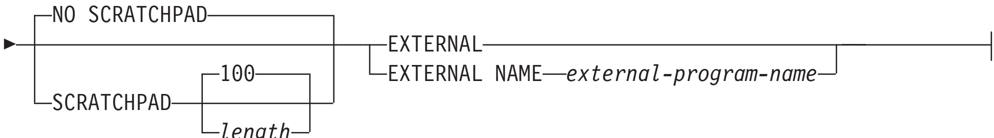
```
|--+-built-in-type------+--------------------------------------|
   '-distinct-type-name-'
```

built-in-type:



option-list:





Notes:

1 This clause and the clauses that follow in the *option-list* can be specified in any order. Each clause may be specified at most once.

Description

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. See "Choosing the Schema and Function Name" on page 310 and "Determining the Uniqueness of Functions in a Schema" on page 311.

(*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. A function can have no input parameters. See "Defining the Parameters" on page 310.

parameter-name
Names the parameter. Although not required, a parameter name can be specified for each parameter. The name cannot be the same as any other *parameter-name* for the function.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 353 for a more complete description of each built-in data type. Some data types are not supported in all languages. See "Attributes of the Arguments Passed to a Routine Program" on page 660 for details on the mapping between the SQL data types and host language data types. The built-in data type specifications that have a correspondence in the language that is being used to write the user-defined function may be specified.

Parameters with a large object (LOB) data type are not supported when PARAMETER STYLE JAVA is specified.

distinct-type-name
Specifies a user-defined distinct type. The length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE DISTINCT TYPE). See "CREATE DISTINCT TYPE" on page 304 for more information.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

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.

RETURNS

Specifies the data type for the result of the function. Consider this clause in conjunction with the optional CAST FROM clause.

data-type2

Specifies the data type of the result.

The same considerations that apply to the data type of input parameters, as described under "data-type1" on page 316, apply to the data type of the result of the function.

AS LOCATOR

Specifies that the function returns a locator to the value rather than the actual value. 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.

data-type3 **CAST FROM** *data-type4*

Specifies the data type of the result of the function (*data-type4*) and the data type in which that result is returned to the invoking statement (*data-type3*). The two data types can be different. For example, for the following definition, the function returns a DOUBLE value, which the database manager converts to a DECIMAL value and then passes to the statement that invoked the function:

```
CREATE FUNCTION SQRT(DECIMAL(15,0))
      RETURNS DECIMAL(15,0)
      CAST FROM DOUBLE
      ...
```

The value of *data-type4* must not be a distinct type , and must be castable to *data-type3*. The value for *data-type3* can be any built-in data type or distinct type. For information on casting data types, see "Casting Between Data Types" on page 54.

AS LOCATOR

Specifies that the external program returns a locator to the value rather than the value. The value represented by this locator is then cast to *data-type3*. Specify AS LOCATOR only if *data-type4* is a LOB data type or a distinct type based on a LOB data type.

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.

C Specifies that the external function is written in C or C++. The database manager will invoke the function using the C language calling conventions.

JAVA

Specifies that the external function is written in Java. The database manager will invoke the function which must be a public static method of the specified Java class.

When LANGUAGE JAVA is specified, the EXTERNAL NAME clause must be specified with a valid *external-java-routine-name*. Do not specify LANGUAGE JAVA when SCRATCHPAD, FINAL CALL, or DBINFO is specified.

PARAMETER STYLE
Specifies the conventions used for passing parameters to and returning a value from the function. See "Parameter Passing for External Routines" on page 653 for details.

DB2SQL
Specifies that in addition to the arguments specified in the invocation of the function, that parameters for indicator variables are associated with each input and return value to allow for null values. PARAMETER STYLE DB2SQL must be specified when LANGUAGE C is also specified. The parameters that are passed between the invoking SQL statement and the function include:

- The first *n* parameters are the input parameters that are specified for the function
- A parameter for the result of the function
- *n* parameters for the indicator variables for the input parameters
- A parameter for the indicator variable for the result
- The SQLSTATE to be returned to DB2
- The qualified name of the function
- The specific name of the function
- The SQL diagnostic string to be returned to DB2

Zero to three additional parameters might also be passed:

- The scratchpad, if SCRATCHPAD is specified
- The call type, if FINAL CALL is specified
- The DBINFO structure, if DBINFO is specified

JAVA
Specifies that the user-defined function uses a convention for passing parameters that conforms to the Java and SQLJ specifications. PARAMETER STYLE JAVA must be specified when LANGUAGE JAVA is also specified and cannot be specified with any other LANGUAGE value.

When PARAMETER STYLE JAVA is specified, do not specify SCRATCHPAD, FINAL CALL, or DBINFO.

SPECIFIC *specific-name*
Specifies a unique name for the function. See "Specifying a Specific Name for a Function" on page 312.

NOT DETERMINISTIC or **DETERMINISTIC**
Specifies whether the function returns the same results each time the function is invoked with the same input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function may not return the same result each time the function is invoked with the same input arguments . The function depends on some state values that affect the results. The database manager uses this information in optimization of SQL statements. An example of a function that is not deterministic is one that generates random numbers.

Some functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks. Specify the DISALLOW PARALLEL clause for these functions.

DETERMINISTIC
Specifies that the function always returns the same result each time the function is invoked with the same input arguments. The database manager uses this information in optimization of SQL statements. An example of a deterministic function is a function that calculates the square root of the input argument.

The database manager does not verify that the function program is consistent with the specification of NOT DETERMINISTIC or DETERMINISTIC.

NO SQL
Specifies that the function cannot execute any SQL statements.

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.

STATIC DISPATCH
This optional clause specifies that at function resolution time that the function should be chosen based on the static (or declared) types of the input parameters of the function.

G DB2 UDB for UWO does not support specification of the STATIC DISPATCH
G clause.

NO DBINFO or **DBINFO**
Specifies whether additional status information is passed when the function is invoked. The default is NO DBINFO.

NO DBINFO
Specifies that no additional information is passed.

DBINFO
Specifies that an additional argument is passed when the function is invoked. The argument is a structure that contains information such as the application run-time authorization ID, the schema name, and identification of the database manager that invoked the function. For details about the argument and its structure, see “Database Information in External Routines (DBINFO)” on page 662, or the sqludf include file.

Do not specify DBINFO when LANGUAGE JAVA is specified.

EXTERNAL ACTION or **NO EXTERNAL ACTION**
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that the database manager does not manage.

Some functions with external actions can receive incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information in optimization of SQL statements.

The database manager does not verify that the function program is consistent with the specification of EXTERNAL ACTION or NO EXTERNAL ACTION.

FENCED
Specifies that the external function runs in an environment that is isolated from the database manager environment.

NO FINAL CALL or **FINAL CALL**
Specifies whether a *final call* is made to the function. A final call enables the function to free any system resources that it has acquired. A final call is useful when the function has been defined with the SCRATCHPAD keyword and the function acquires system resource and stores them in the scratchpad. The default is NO FINAL CALL.

NO FINAL CALL
Specifies that a final call is not made to the function. The function does not receive an additional argument that specifies the type of call.

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.

Do not specify FINAL CALL when LANGUAGE JAVA is specified.

The types of calls are:

First call
Specifies that the first call to the function for this reference to the function in this SQL statement. A first call is a normal call. SQL arguments are passed and the function is expected to return a result.

Normal call
Specifies that SQL arguments are passed and the function is expected to return a result.

Final call
Specifies that the last call to the function to enable the function to free resources. A final call is not a normal call. A final call occurs at these times:

- *End of statement:* When the cursor is closed for cursor-oriented statements, or the execution of the statement has completed.
- *End of a parallel task:* When the function is executed by parallel tasks.
- *End of transaction:* When normal end of statement processing does not occur. For example, the logic of an application, for some reason, bypasses closing the cursor.

If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made when the cursor is closed or the application ends. If a commit occurs at the end of a parallel task, a final call is made regardless of whether a cursor defined as WITH HOLD is open.

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.

ALLOW PARALLEL or **DISALLOW PARALLEL**

For a single reference to the function, specifies whether parallelism can be used when the function is invoked. Although parallelism can be used for most scalar functions, some functions such as those that depend on a single copy of the scratchpad cannot be invoked with parallel tasks. Consider these characteristics when determining which clause to use:

- If all invocations of the function are completely independent from one another, specify ALLOW PARALLEL.
- If each invocation of the function updates the scratchpad, providing values that are of interest to the next invocation, such as incrementing a counter, specify DISALLOW PARALLEL.
- If the scratchpad is used only so that some expensive initialization processing is performed a minimal number of times, specify ALLOW PARALLEL.

The default is DISALLOW PARALLEL, if one or more of the following clauses is specified: NOT DETERMINISTIC, EXTERNAL ACTION, FINAL CALL, SCRATCHPAD. Otherwise, ALLOW PARALLEL is the default.

ALLOW PARALLEL

Specifies that the database manager can consider parallelism for the function. The database manager is not required to use parallelism on the SQL statement that invokes the function. Existing restrictions on parallelism apply.

See NOT DETERMINISTIC, EXTERNAL ACTION, SCRATCHPAD, and FINAL CALL for considerations when specifying ALLOW PARALLEL.

DISALLOW PARALLEL

Specifies that the database manager must not use parallelism for the function.

NO SCRATCHPAD or **SCRATCHPAD**

Specifies whether the database manager provides a scratchpad for the function. It is strongly recommended that external functions be reentrant. A scratchpad provides an area for the function to save information from one invocation to the next. The default is NO SCRATCHPAD.

NO SCRATCHPAD

Specifies that a scratchpad is not allocated and passed to the function.

SCRATCHPAD *length*

Specifies that when the function is invoked for the first time, the database manager allocates memory for a scratchpad. A scratchpad has the following characteristics:

- *length* must be between 1 and 32767. The default value is 100 bytes.

- The database manager initializes the scratchpad to all binary zeros (X'00''s).
- The scope of a scratchpad is the SQL statement. For each reference to the function in an SQL statement, there is one scratchpad. For example, assuming that function UDFX was defined with the SCRATCHPAD keyword, three scratchpads are allocated for the three references to UDFX in the following SQL statement:

```
SELECT A, UDFX(A)
  FROM TABLEB
 WHERE UDFX(A) > 103
    OR UDFX(A) < 19;
```

  If the function is run under parallel tasks, one scratchpad is allocated for each parallel task of each reference to the function in the SQL statement. This can lead to unpredictable results. For example, if a function uses the scratchpad to count the number of times that it is invoked, the count reflects the number of invocations done by the parallel task and not the SQL statement. Specify the DISALLOW PARALLEL clause for functions that will not work correctly with parallelism.
- The scratchpad is persistent. The database manager preserves its content from one invocation of the function to the next. Any changes that the function makes to the scratchpad on one call are still there on the next call. The database manager initializes the scratchpads when it begins to execute an SQL statement. The database manager does not reset scratchpads when a correlated subquery begins to execute.
- The scratchpad can be a central point for the system resources that the function acquires. If the function acquires system resources, specify FINAL CALL to ensure that the database manager calls the function one more time so that the function can free those system resources.

Each time the function invoked, the database manager passes an additional argument to the function that contains the address of the scratchpad.

Do not specify SCRATCHPAD when LANGUAGE JAVA is specified.

EXTERNAL

Specifies that the CREATE FUNCTION statement is being used to define a new function based on code written in an external programming language.

If the NAME clause is not specified, 'NAME *function-name*' is implicit. In this case, *function-name* must not be longer than 8 characters. The NAME clause is required for a LANGUAGE JAVA function since the default name is not valid for a Java function.

NAME *external-program-name*

Specifies the program that will be executed when the function is invoked. The executable form of the external program need not exist when the CREATE FUNCTION statement is executed. However, it must exist at the current server when the function is invoked.

If a JAR is referenced then the JAR must exist when the function is created.

Notes

General Considerations for Defining User-defined Functions: See "CREATE FUNCTION" on page 310 for general information on defining user-defined functions.

Owner Privileges: The owner is authorized to execute the function with the ability to grant these privileges to others. See "GRANT (Function or Procedure Privileges)" on page 412. For more information on ownership of the object, see "Authorization, Privileges and Object Ownership" on page 27.

Language Considerations: A C program must be written to run as a subroutine.

For information on creating the programs for a function, see Appendix L, "Coding Programs for use by External Routines" on page 653.

Examples

Example 1: Assume an external function program in C is needed that implements the following logic:

```
rslt   = 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
```

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 defines the function, and lets the database manager generate a specific name for the function.

```
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 (Sourced)

The CREATE FUNCTION (Sourced) statement is used to define a user-defined function that is based on an 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 as least one of the following:

- The privilege to create in the schema
- Administrative authority

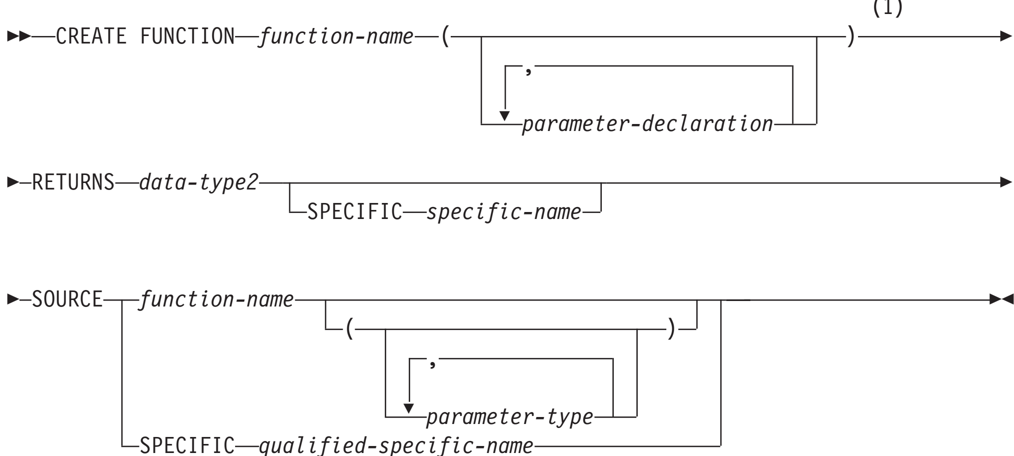
The authorization ID must have one of the following privileges:

- The EXECUTE privilege for the function identified in the SOURCE clause
- Administrative authority

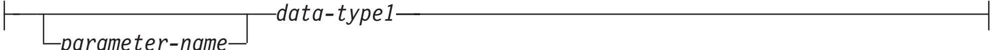
For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Administrative authority.

Syntax



parameter-declaration:



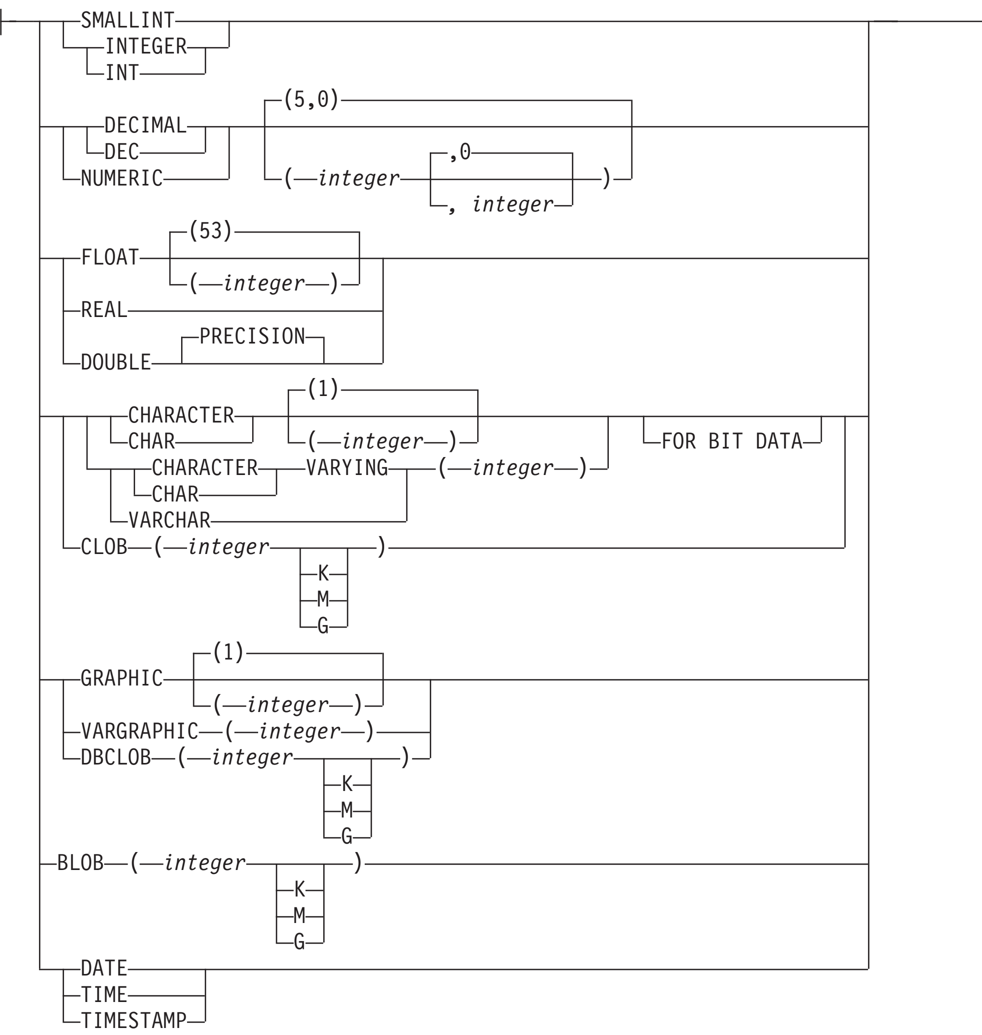
Notes:

1 RETURNS, SPECIFIC, and SOURCE can be specified in any order.

data-type1, data-type2, data-type3:

```
|--+-built-in-type------+--------------------------------|
   '-distinct-type-name-'
```

built-in-type:



parameter-type:

Notes:

1 Empty parentheses are allowed for some data types specified in this context.

Description

function-name
Names the user-defined function. If *function-name* is not qualified, it is

implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. See “Choosing the Schema and Function Name” on page 310 and “Determining the Uniqueness of Functions in a Schema” on page 311.

(*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. A function can have no input parameters. See “Defining the Parameters” on page 310.

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 data type of the parameter. The data type can be a built-in data type or a distinct type.

Any valid SQL data type may be used provided it is castable to the type of the corresponding parameter of the function identifed in the SOURCE clause (for information see “Casting Between Data Types” on page 54). 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” on page 330.

built-in-type
The data type of the input parameter is a built-in data type. See “CREATE TABLE” on page 353 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 type of the distinct type (those specified on CREATE DISTINCT TYPE). See “CREATE DISTINCT TYPE” on page 304 for more information.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

RETURNS
Specifies the result of the function.

data-type2
Specifies the data type of the result.

Any built-in data type or distinct type can be specified. The data type of the final result of the source function must match or be castable to the result of the sourced function. For information on casting data types, see “Casting Between Data Types” on page 54. However, this checking does not guarantee that an error will not occur when this new function is invoked. See “Considerations for Invoking a Sourced User-defined Function” on page 330.

SPECIFIC *specific-name*
Specifies a unique name for the function. See “Specifying a Specific Name for a Function” on page 312.

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 one of the following types of functions:

- a function that was defined with a CREATE FUNCTION statement
- a cast function that was generated by a CREATE DISTINCT TYPE statement
- a built-in function

If the source function is not a built-in function, the particular function can be identified by its name, function signature, or specific name.

If the source function is a built-in function, the SOURCE clause must include a function signature for the built-in function. The source function must not be any of the following built-in functions (if a particular syntax is shown, only the indicated form cannot be specified) :

- BLOB when more than one argument is specified
- CHAR when more than one argument is specified
- CLOB when more than one argument is specified
- COUNT(*)
- COUNT_BIG(*)
- COALESCE
- DBCLOB when more than one argument is specified
- DECIMAL when more than one argument is specified
- GRAPHIC when more than one argument is specified
- MAX
- MIN
- NULLIF
- TRANSLATE when more than one argument is specified
- VARCHAR when more than one argument is specified
- VARGRAPHIC when more than one argument is specified
- VALUE

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.

If an unqualified *function-name* is specified, the SQL path is used to locate the function. The first schema that has only one function with this name on which the user has EXECUTE authority is selected. An error is returned if a function is not found, or a schema is encountered that has more than one function with this name.

function-name (parameter-type,...)

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. 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.

Empty parentheses are allowed for some data types specified in this context. 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 parentheses because its parameter value indicates a specific data type (REAL or DOUBLE).
- If a specific value for a length, precision, or scale attribute is specified, the value must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement. If the data type is FLOAT, the precision does not have to exactly match the value that was specified because matching is based on the data type (REAL or DOUBLE).
- If length, precision, or scale is not explicitly specified, and empty parentheses are not specified, the default attributes of the data type are implied. The implicit length must exactly match the value that was specified (implicitly or explicitly) in the CREATE FUNCTION statement.

AS LOCATOR
: Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or distinct type based on a LOB.

SPECIFIC *qualified-specific-name*
: Identifies the function to be used as the source function by its specific name. The *qualified-specific-name* must identify a specific function that exists in the specified or implicit schema.

Notes

General Considerations for Defining User-defined Functions: See "CREATE FUNCTION" on page 310 for general information on defining user-defined functions.

Owner Privileges: The owner is authorized to execute the function.

- If the underlying function is a user-defined function, and the owner is authorized with the grant option to execute the underlying function, then the privilege on the new function includes the grant option. Otherwise, the owner can execute the new function but cannot grant others the privilege to do so.
- If the underlying function is a built-in function, the owner is authorized with the grant option to execute the underlying built-in function and the privilege on the new function includes the grant option.

See "GRANT (Function or Procedure Privileges)" on page 412. For more information on ownership of the object, see "Authorization, Privileges and Object Ownership" on page 27.

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 defined by the options specified implicitly or explicitly when the external scalar function was created. 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.

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)
```

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)

This statement is used to define a user-defined SQL scalar function at the current server. A scalar function returns a single value each time it is invoked.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

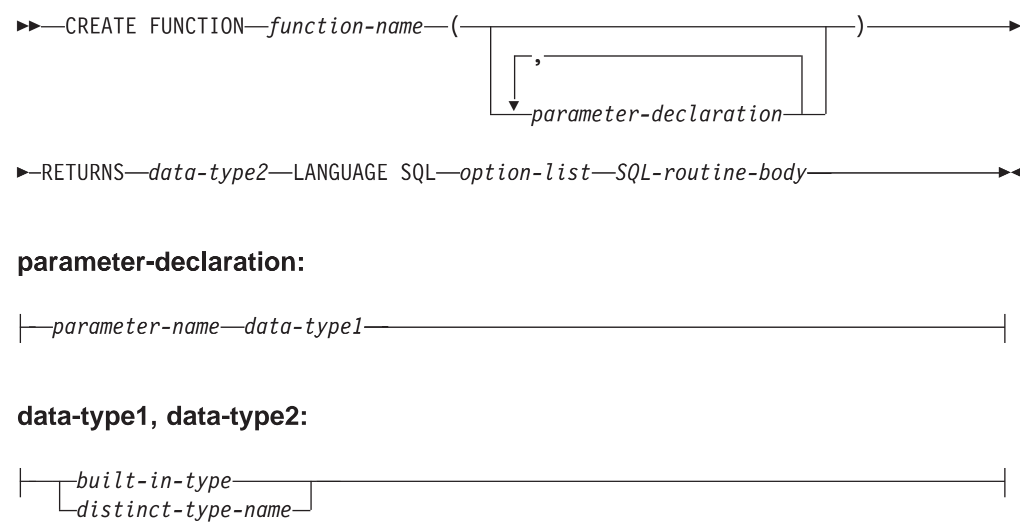
The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority

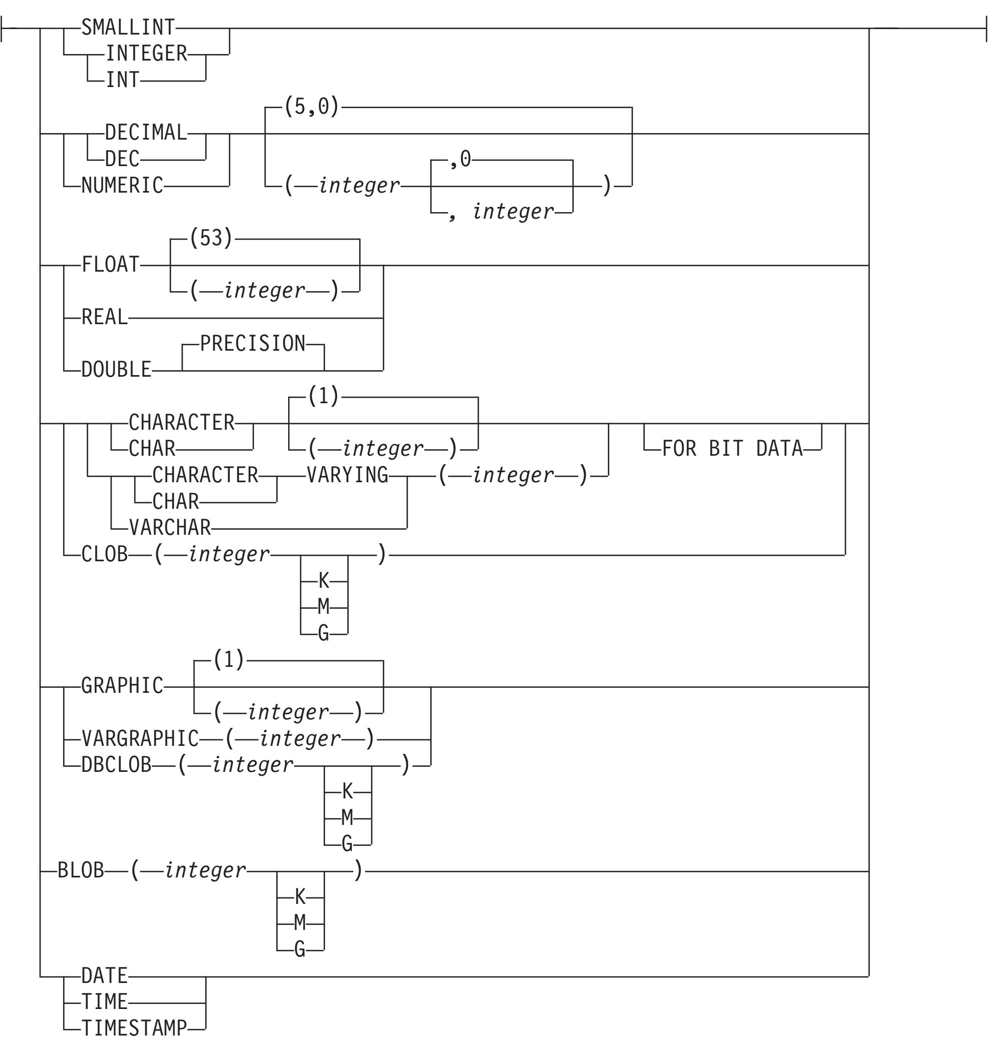
For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Administrative authority.

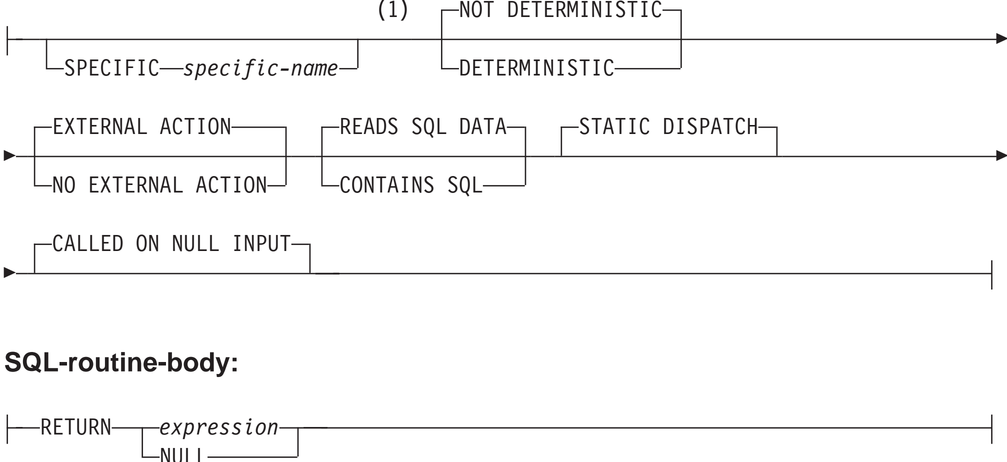
Syntax



built-in-type:



option-list:



SQL-routine-body:

```
├──RETURN──┬─expression─┬─────────────────────────────────────┤
           └─NULL───────┘
```

Notes:

1 This clause and the clauses that follow in the *option-list* can be specified in any order. Each clause may be specified at most once.

Description

function-name
Names the user-defined function. If *function-name* is not qualified, it is implicitly qualified with the default schema name. The schema must be a valid schema name for functions, and the unqualified function name must not be any of the reserved function names. The same name can be used for more than one function in the same schema if the function signature of each function is unique. See "Choosing the Schema and Function Name" on page 310 and "Determining the Uniqueness of Functions in a Schema" on page 311.

(*parameter-declaration,...***)**
Specifies the number of input parameters of the function and the name and data type of each parameter. Each *parameter-declaration* is an input parameter for the function. A maximum of 90 parameters can be specified. A function can have no input parameters. See "Defining the Parameters" on page 310.

parameter-name
Specifies the name of the input parameter. A name must be specified for each parameter. The name is used to refer to the parameter within the body of the function. Each name in the parameter list must not be the same as any other name.

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type or a distinct type.

built-in-type
The data type of the input parameter is a built-in data type. See "CREATE TABLE" on page 353 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 type of the distinct type (those specified on CREATE DISTINCT TYPE). See "CREATE DISTINCT TYPE" on page 304 for more information.

If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

RETURNS
Specifies the result of the function.

data-type2
Specifies the data type of the result.

Similar considerations that apply to the data type of the input parameters, as described in "Defining the Parameters" on page 310, apply to the data type of the result of the function.

LANGUAGE SQL
Specifies the function is written exclusively in SQL.

SPECIFIC *specific-name*
Provides a unique name for the function. See "Specifying a Specific Name for a Function" on page 312.

NOT DETERMINISTIC or **DETERMINISTIC**
Specifies whether the function returns the same results for identical input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function may not return the same result each time the function is invoked with the same input arguments. The function depends on some state values that affect the results. The database manager uses this information in optimization of SQL statements. An example of a function that is not deterministic is one that generates random numbers.

Some functions that are not deterministic can receive incorrect results if the function is executed by parallel tasks.

DETERMINISTIC
Specifies that the function always returns the same result each time the function is invoked with the same input arguments. The database manager uses this information in optimization of SQL statements. An example of a deterministic function is a function that calculates the square root of the input argument.

The SQL routine body must be consistent with the specification of the implicit or explicit specification of DETERMINISTIC or NOT DETERMINISTIC. A function defined as DETERMINISTIC must not invoke another function defined as NOT DETERMINISTIC, or reference a special register. For example, an SQL function could invoke the RAND built-in function in the RETURN statement. DETERMINISTIC must be specified in the CREATE FUNCTION statement for such a function.

EXTERNAL ACTION or **NO EXTERNAL ACTION**
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function can take an action that changes the state of an object that the database manager does not manage.

Some functions with external actions can receive incorrect results if parallel tasks execute the function. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information in optimization of SQL statements.

EXTERNAL ACTION must be implicitly or explicitly specified if the SQL routine body invokes a function that is defined with EXTERNAL ACTION.

READS SQL DATA or **CONTAINS SQL**
Specifies the classification of SQL statements the function can execute. The database manager verifies that the SQL issued by the function is consistent with this specification. For classification of each statement, see “SQL Statement Data Access Classification for Routines” on page 518. The default is READS SQL DATA.

READS SQL DATA
Specifies that that function can execute statements with a data access indication of READS SQL DATA or CONTAINS SQL. The function cannot execute SQL statements that modify data.

CONTAINS SQL
Specifies that the function can only execute statements with a data access indication of CONTAINS SQL. The function cannot execute any SQL statements that read or modify data. For example, READS SQL DATA (or MODIFIES SQL DATA) must be specified if the body of the SQL function contains a subselect, or invokes a function that can read data.

CALLED ON NULL INPUT
Specifies that the function is called regardless of whether any of the input arguments is null, making the function responsible for testing for null argument values. The function can return a null or nonnull value.

STATIC DISPATCH
This optional clause specifies that at function resolution time that the function should be chosen based on the static (or declared) types of the input parameters of the function.

G G DB2 UDB for UWO does not support specification of the STATIC DISPATCH clause.

RETURN
Specifies the return value of the function. Parameter names can be referenced in the RETURN statement. Parameter names may be qualified by the function name to avoid ambiguous references.

expression
Specifies the expression to be returned for the function. The result data type of the expression must be assignable (using storage assignment rules) to the data type defined in the RETURNS clause. See "Storage Assignment" on page 60.

The expression can contain one or more of the following:

- Function (either user-defined or built-in)
- Expression enclosed in parenthesis
- Constant
- Special register
- Labeled duration
- CASE expression
- CAST specification

The expression cannot include a column name or a host variable.

The expression cannot contain a recursive invocation of itself or to another function that invokes it, since such a function could not exist to be referred to.

NULL
Specifies that the function returns a null value of the data type defined in the RETURNS clause.

Notes

General Considerations for Defining User-defined Functions: See "CREATE FUNCTION" on page 310 for general information on defining user-defined functions.

Owner Privileges: The owner is authorized to execute the function. The EXECUTE privilege can be granted to others only if the owner has the authority to grant the

EXECUTE privilege on every user-defined function referenced in the RETURN statement of the function body. See "GRANT (Function or Procedure Privileges)" on page 412. For more information on ownership of the object, see "Authorization, Privileges and Object Ownership" on page 27.

SQL path and Function Resolution: Resolution of function invocations inside the function body is done according to the SQL path that is in effect for the CREATE FUNCTION statement and does not change after the function is created.

References to Datetime Special Registers: If an SQL function contains multiple references to any of the date or time special registers, all references return values corresponding to the same timestamp. The value of that timestamp is the value that would be returned by referencing the CURRENT TIMESTAMP special register in the statement that invoked the function.

Examples

Example 1: 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 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

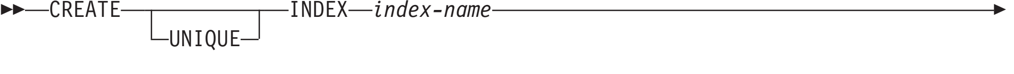
The privileges held by the authorization ID of the statement must include at least one of the following:

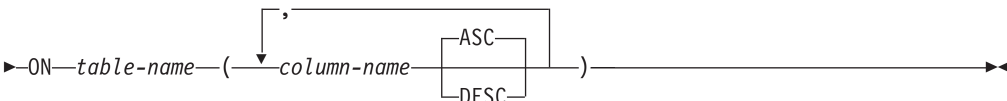
- The privilege to create in the schema
- Administrative authority.

The privileges held by the authorization ID of the statement must include at least one of the following:

- The INDEX privilege for the table
- 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 and an error is returned.

  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.

index-name
: Names the index. The name, including the implicit or explicit qualifier, must not identify an index, table, view, or alias that already exists at the current server. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT.

ON *table-name*
: Identifies the table on which the index is to be created. The name must identify a base table that exists at the current server, but it must not identify a catalog table.

(*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. A column name must not be specified more than once, and must not identify a LOB column or a column defined as a distinct type which is based on a LOB data type. The number of specified columns must not exceed 16 and the sum of their length attributes must not exceed 255. See Table 41 on page 511 for more information. Note that this length can be reduced by system overhead which varies according to the data type of the column and whether it is nullable. See "Byte Counts" on page 366 for more information on overhead affecting this limit.

ASC
Puts the index entries in ascending order by the column. This is the default.

DESC
Puts the index entries in descending order by the column.

Ordering is performed in accordance with the comparison rules described in Chapter 2, "Language Elements" on page 29. The null value is higher than all other values.

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.

Owner Privileges: There are no specific privileges on an index. For more information on ownership of the object, see "Authorization, Privileges and Object Ownership" on page 27.

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 341.

- 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 348.

Notes

Special Registers in Procedures: The settings of the special registers of the caller are inherited by the procedure on invocation and restored upon return to the caller. Special registers may be changed within a procedure, but these changes do not affect the caller.

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

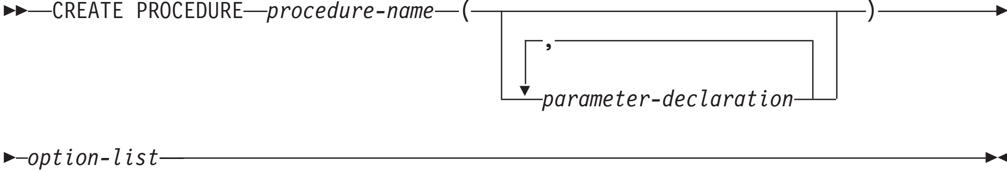
The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority.

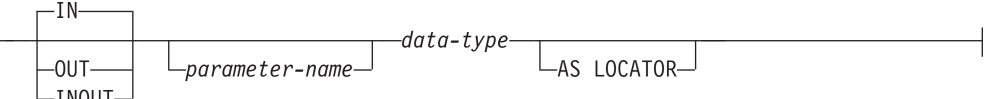
For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Administrative authority.

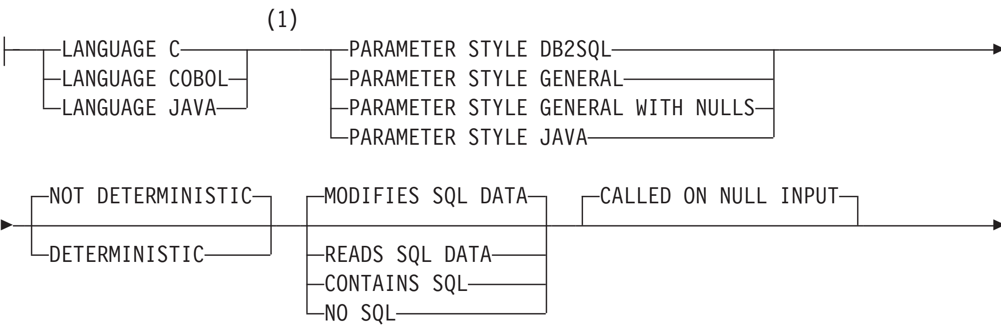
Syntax



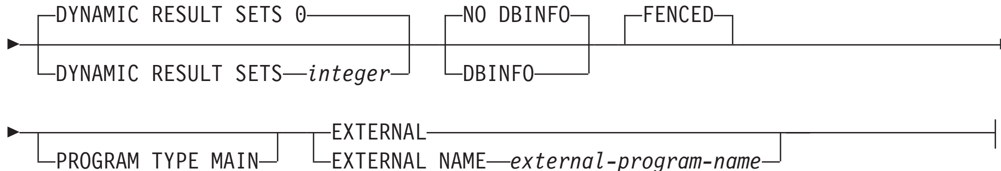
parameter-declaration:



option-list:



CREATE PROCEDURE (External)



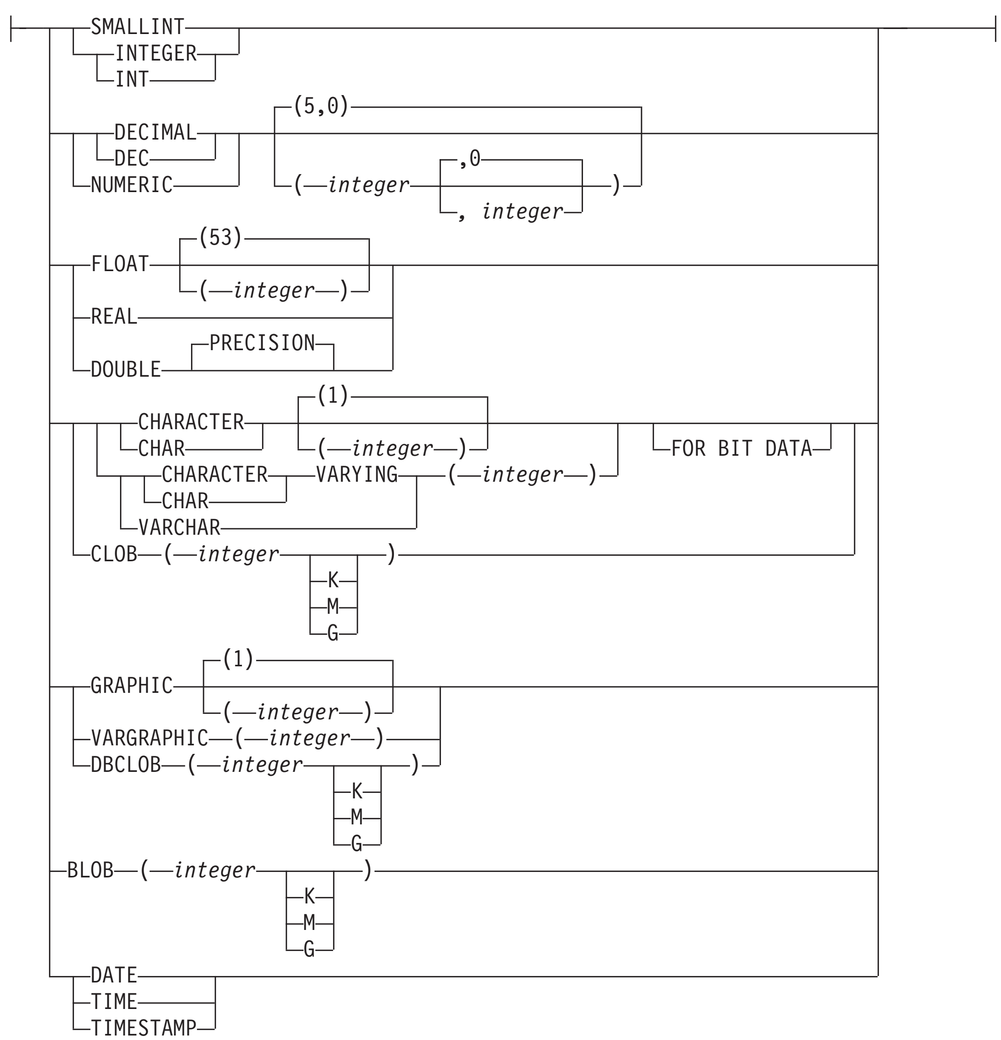
Notes:

1 The clauses in the option-list can be specified in any order.

data-type:

built-in-type
distinct-type-name

built-in-type:



Description

procedure-name
Names the procedure. The name, including the implicit or explicit qualifier, must not identify a procedure that already exists at the current server. If a qualified procedure name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 3).

(*parameter-declaration***,...)**
Specifies the number of parameters of the procedure, the data type of each parameter, and, optionally, the name of each parameter. A parameter for a procedure can be used only for input, only for output, or for both input and output. A maximum of 90 parameters can be specified. See Appendix A, "SQL Limits" on page 509 for more details on limits.

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. The default is IN.

OUT
Identifies the parameter as an output parameter that is returned by the procedure.

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.

G In DB2 UDB for UWO, *parameter-name*s must be specified.

data-type
Specifies the data type of the parameter.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" on page 353 for a more complete description of each built-in data type.

Parameters with a large object (LOB) data type are not supported when PARAMETER STYLE JAVA is specified.

distinct-type-name
Specifies a distinct type. Any length, precision, scale, or encoding scheme attributes for the parameter are those of the source type of the distinct type as specified using "CREATE DISTINCT TYPE" on page 304.

G In DB2 UDB for UWO, distinct types are not supported in procedures.

See "Attributes of the Arguments Passed to a Routine Program" on page 660 for details on the mapping between the SQL data types and host language data types. Some data types are not supported in all languages.

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the procedure instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type. Passing locators instead of values can result in fewer bytes being passed to and from the procedure, especially when the value of the parameter is very large.

G In DB2 UDB for UWO the AS LOCATOR clause is not supported.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the procedure body is written. All programs must be designed to run in the server's environment.

C The external program is written in C or C++. The database manager will call the procedure using the C language calling conventions.

COBOL
The external program is written in COBOL. The database manager will call the procedure using the COBOL language calling conventions.

JAVA
The external program is written in Java. The database manager will call the procedure using the Java language calling conventions. The procedure must be a public static method of the specified Java class.

PARAMETER STYLE
Specifies the conventions used for passing parameters to and returning the values from procedures. See "Parameter Passing for External Routines" on page 653 for details.

DB2SQL
Specifies that in addition to the parameters on the CALL statement, several additional parameters are passed to the procedure. The following arguments are passed to the procedure:

- The first *n* parameters are the parameters that are specified on the CREATE PROCEDURE statement.
- *n* parameters for indicator variables for the parameters.
- The SQLSTATE to be returned.
- The qualified name of the procedure.
- The specific name of the procedure.
- The SQL diagnostic string to be returned.
- If DBINFO is specified, the DBINFO structure.

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. Arguments to procedures defined with this parameter style cannot be null.

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.

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.

PARAMETER STYLE JAVA can only be used with LANGUAGE JAVA.

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. The default is NOT DETERMINISTIC.

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.

NOT 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.

MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or **NO SQL**
Specifies which SQL statements, if any, may be executed in the procedure or any routine called from this procedure. The default is MODIFIES SQL DATA. For data access classification of each statement, see "SQL Statement Data Access Classification for Routines" on page 518.

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

READS SQL DATA
Specifies that the procedure can execute statements with a data access classification of READS SQL DATA or CONTAINS SQL.

CONTAINS SQL
Specifies that the procedure can only execute statements with a data access classification of CONTAINS SQL.

NO SQL
Specifies that the procedure cannot execute any SQL statements.

CALLED ON NULL INPUT
Specifies that the procedure will be called if any, or all, parameter values are null.

DYNAMIC RESULT SETS *integer*
Specifies the maximum number of result sets that can be returned from the procedure. The minimum value for *integer* is zero and the maximum value is 32767. The default is DYNAMIC RESULT SETS 0.

NO DBINFO or **DBINFO**
Specifies whether additional status information is passed to the procedure when it is invoked. The default is NO DBINFO.

NO DBINFO
Additional information is not passed.

DBINFO
An additional argument is passed when the procedure is invoked.

The argument is a structure that contains information such as the name of the current server, the application run-time authorization ID and

identification of the version and release of the database manager that invoked the procedure. See “Database Information in External Routines (DBINFO)” on page 662 for further details.

DBINFO can be specified only if PARAMETER STYLE DB2SQL is specified.

FENCED
Specifies that the procedure runs in an environment that is isolated from the database manager environment.

PROGRAM TYPE MAIN
Specifies that the procedure executes as a main routine. PROGRAM TYPE MAIN is only valid for LANGUAGE C or COBOL.

G The default when PROGRAM TYPE MAIN is not specified is product specific.

EXTERNAL
Specifies that the CREATE PROCEDURE statement is being used to define a new procedure based on code written in an external programming language.

If NAME clause is not specified "NAME *procedure-name*" is assumed. In this case, *procedure-name* must not be longer than 8 characters. The NAME clause is required for a LANGUAGE JAVA procedure since the default name is not valid for a Java procedure.

NAME *external-program-name*
Specifies the program that will be executed when the procedure is called by the CALL statement. The executable form of the external program need not exist when the CREATE PROCEDURE statement is executed. However, it must exist at the current server when the procedure is called.

If a JAR is referenced then the JAR must exist when the procedure is created.

Notes

General Considerations for Defining Procedures: See “CREATE PROCEDURE” on page 340 for general information on defining procedures.

Language Considerations: For information needed to create the programs for a procedure, see Appendix L, “Coding Programs for use by External Routines” on page 653.

Owner Privileges: The owner is authorized to call the procedure and grant others the privilege to call the procedure. See “GRANT (Function or Procedure Privileges)” on page 412. For more information on ownership of the object, see “Authorization, Privileges and Object Ownership” on page 27.

Examples

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)

The CREATE PROCEDURE (SQL) statement defines 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.

In DB2 for OS/390, this statement can only be dynamically prepared and processed using the IBM Stored Procedure Builder, JCL, or the DB2 UDB for z/OS and OS/390 SQL procedure processor (DSNTPSMP). See the product *Application Prgramming and SQL Guide* for more information.

Authorization

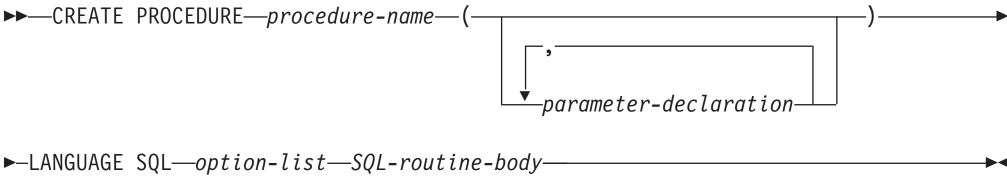
The privileges held by the authorization ID of the statement must include as least one of the following:

- The privilege to create in the schema
- Administrative authority.

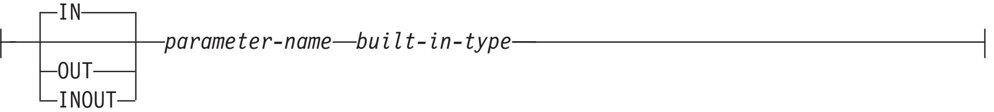
For each distinct type referenced in the statement, the authorization ID of the statement must include at least one of the following:

- The USAGE privilege for the distinct type
- Ownership of the distinct type
- Administrative authority.

Syntax

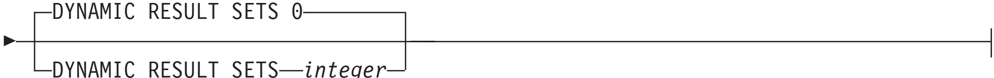


parameter-declaration:



option-list:

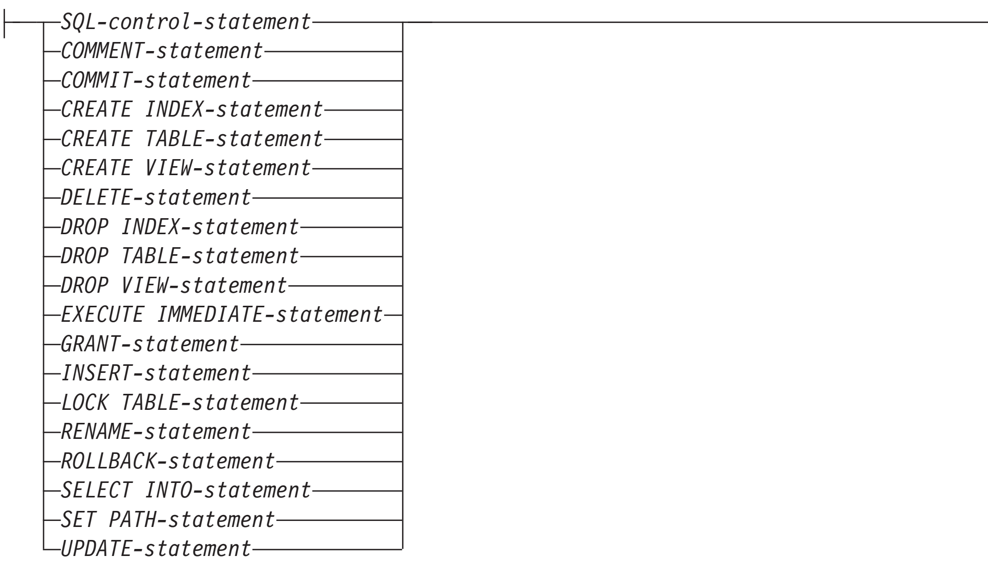
```
   ┌─NOT DETERMINISTIC─┐ (1)  ┌─MODIFIES SQL DATA─┐  ┌─CALLED ON NULL INPUT─┐
├──┼───────────────────┼──────┼───────────────────┼──┴──────────────────────┴──►
   └─DETERMINISTIC─────┘      ├─READS SQL DATA────┤
                              └─CONTAINS SQL──────┘
```



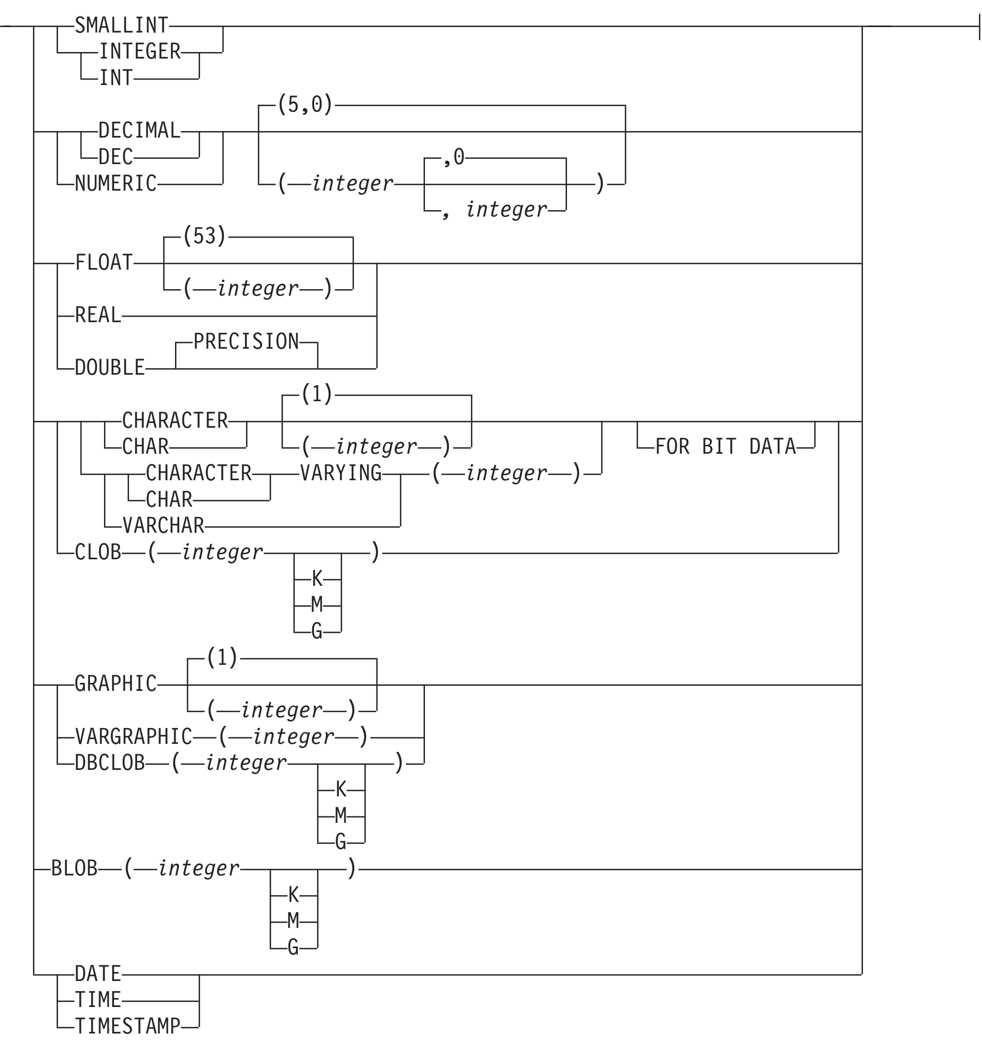
Notes:

1 The clauses in the option-list can be specified in any order.

SQL-routine-body:



built-in-type:



Description

procedure-name
: Names the procedure. The name, including the implicit or explicit qualifier, must not identify a procedure that already exists at the current server. If a qualified procedure name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 3).

(*parameter-declaration***,...)**
: Specifies the number of parameters of the procedure, the data type of each parameter, and the name of each parameter. A parameter for a procedure can be used for input only, for output only, of for both input and output. A maximum of 90 parameters can be specified. See Appendix A, "SQL Limits" on page 509 for more details on limits.

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. The default is IN.

OUT
: Identifies the parameter as an output parameter that is returned by the procedure. If the parameter is not set within the procedure, the null value is returned.

INOUT
Identifies the parameter as both an input and output parameter for the procedure. If the parameter is not set within the procedure, its input value is returned.

parameter-name
Names the parameter for use as an SQL variable. The name cannot be the same as any other *parameter-name* for the procedure.

built-in-type
Specifies the data type of the parameter.

LANGUAGE SQL
Specifies that this procedure is written exclusively in SQL.

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. The default is NOT DETERMINISTIC.

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.

NOT 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.

MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL
Specifies the classification of SQL statements the procedure can execute. The default is MODIFIES SQL DATA. For data access classification of each statement, see "SQL Statement Data Access Classification for Routines" on page 518.

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

READS SQL DATA
Specifies that the procedure can execute statements with a data access classification of READS SQL DATA or CONTAINS SQL.

CONTAINS SQL
Specifies that the procedure can only execute statements with a data access classification of CONTAINS SQL.

CALLED ON NULL INPUT
Specifies that the procedure will be called if any, or all, parameter values are null.

DYNAMIC RESULT SETS *integer*
Indicates the upper bound of returned result sets for the procedure. The default number of DYNAMIC RESULT SETS is 0. The minimum value for *integer* is zero and the maximum value is 32767. The default is DYNAMIC RESULT SETS 0.

SQL-routine-body
Specifies the statements that define the body of the SQL procedure. Multiple

SQL procedure statements may be specified within a compound statement or other SQL control statements. See Chapter 6, "SQL Control Statements" on page 477 for more information.

Notes

General Considerations for Defining Procedures: See "CREATE PROCEDURE" on page 340 for general information on defining procedures.

Owner Privileges: The owner is authorized to call the procedure and grant others the privilege to call the procedure. See "GRANT (Function or Procedure Privileges)" on page 412. For more information on ownership of the object, see "Authorization, Privileges and Object Ownership" on page 27.

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.

Examples

Example 1: 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 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 its 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 privilege to create in the schema
- 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 on the table
- The REFERENCES privilege on each column of the specified parent key
- Ownership of the table
- Administrative authority.

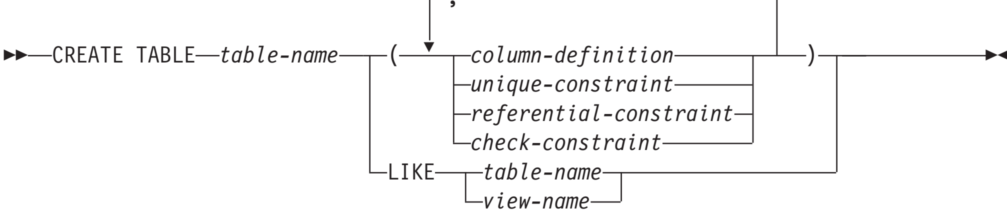
If the LIKE clause is specified, the privileges held by the authorization ID of the statement must include at least one of the following on the table or view specified in the LIKE clause:

- The SELECT privilege for the table or view
- Ownership of the table or view
- Administrative authority.

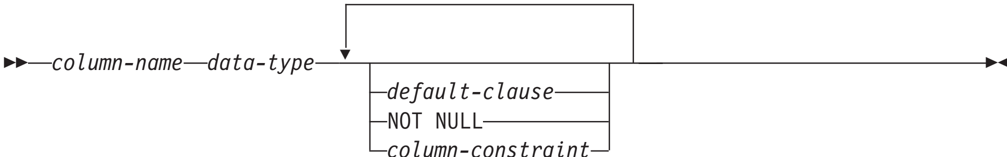
If a distinct type is referenced, the privileges held by the authorization ID of the statement must include at least one of the following:

- The USAGE privilege on the distinct type
- Ownership of the distinct type
- Administrative authority.

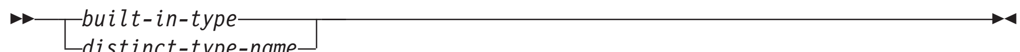
Syntax



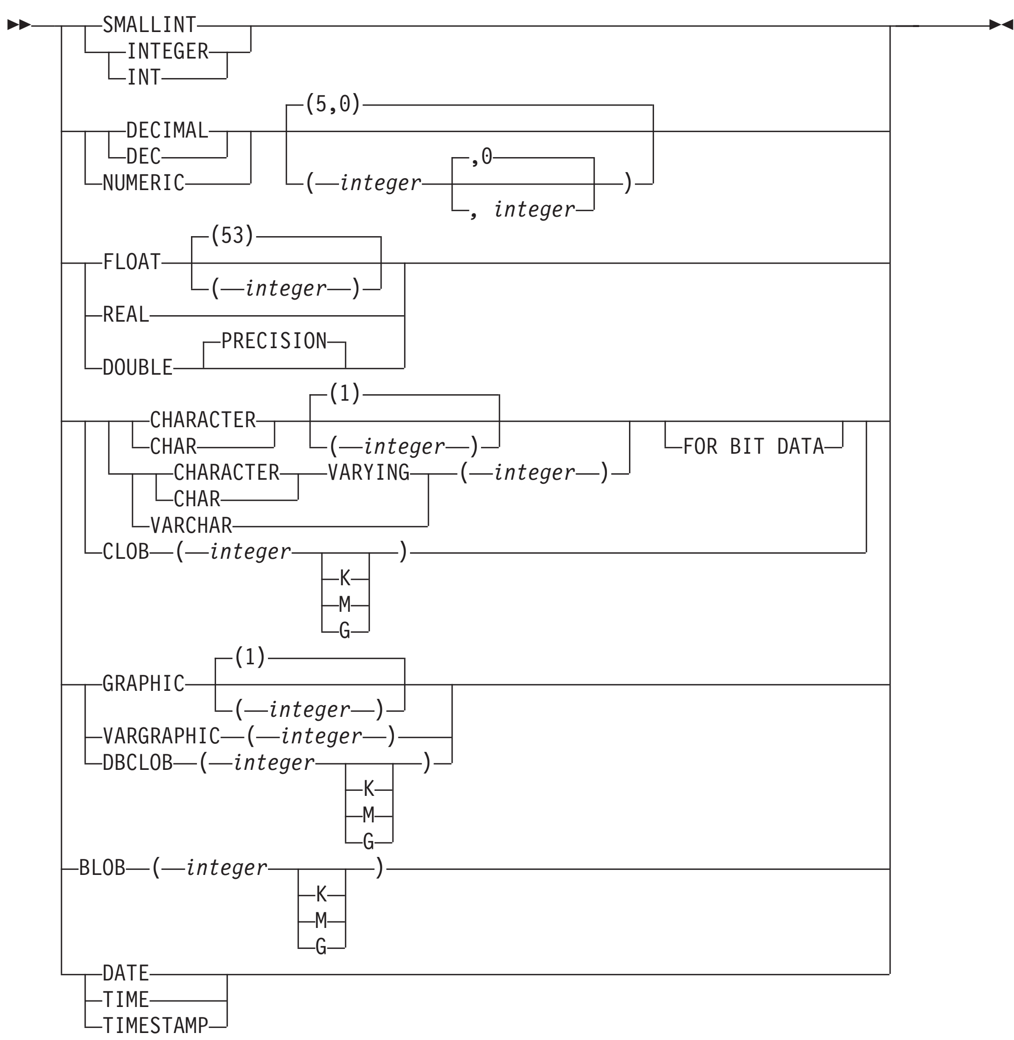
column-definition:



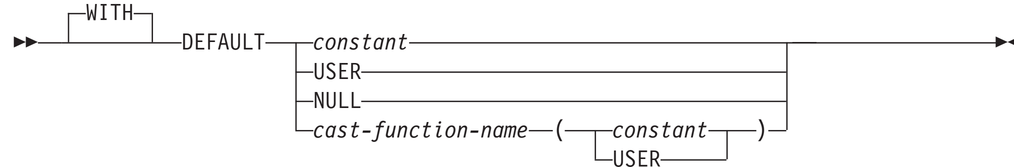
data-type:



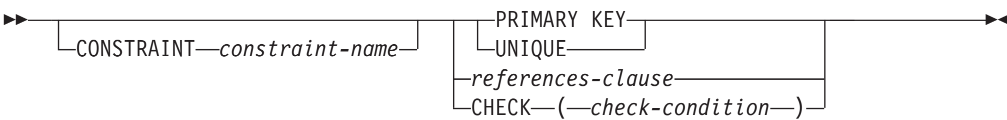
built-in-type:



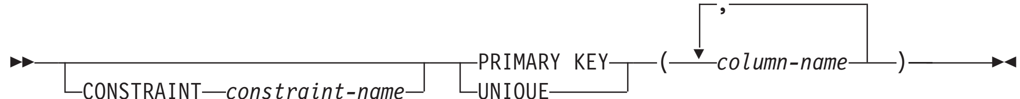
default-clause:



column-constraint:

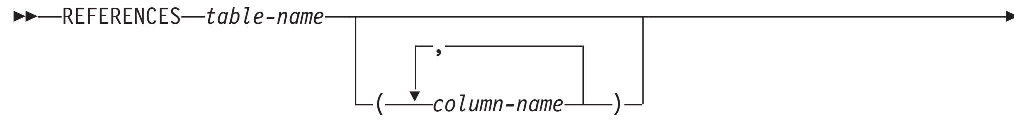


unique-constraint:



referential-constraint:

references-clause:



check-constraint:

Description

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify an alias, index, table or view that already exists at the current server.

column-definition

Defines the attributes of a column. There must be at least one column definition and no more than 750 [57] columns for the table. See Table 41 on page 511 for more information.

column-definition

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.

data-type
Specifies the data type of the column.

built-in-type
For the built-in types, use:

SMALLINT
For a small integer.

INTEGER or **INT**
For a large 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). The scale of the number can range from 0 to the precision of the number.

DECIMAL(*p*) can be used for DECIMAL(*p*,0), and DECIMAL for DECIMAL(5,0).

NUMERIC(*integer, integer***)**
NUMERIC(*integer***)**
NUMERIC
G G For a zoned decimal number in DB2 UDB for iSeries and a packed decimal number in DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO. 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). The scale of the number can range from 0 to the precision of the number.

NUMERIC(*p*) can be used for NUMERIC(*p*,0), and NUMERIC for NUMERIC(5,0).

FLOAT
For a double-precision floating-point number.

57. This value is 1 less if the table is a dependent table.

FLOAT(*integer*)
For a single- or double-precision floating-point number, depending on the value of the integer. The value of the integer must be in the range 1 through 53. The values 1 through *n* indicate single-precision, and the
G values *n* + 1 through 53 indicate double-precision. In DB2 UDB for z/OS
G and OS/390, *n* is 21; in DB2 UDB for UWO and DB2 UDB for iSeries, *n* is
G 24. For portability across operating systems, when specifying a floating-point data type, use REAL or DOUBLE instead of FLOAT.

REAL
For single-precision floating point.

DOUBLE PRECISION
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 to 254. See Table 39 on page 510 for more information.

CHARACTER VARYING(*integer*) or **CHAR VARYING(*integer*)**
VARCHAR(*integer*)
For a varying-length character string of maximum length *integer*. The integer can range from 1 to 32 672. See Table 39 on page 510 for more information.

For the restrictions that apply to the use of VARCHAR strings longer than 255, see "Limitations on Use of Strings" on page 48.

FOR BIT DATA
Indicates that the values of the CHAR or VARCHAR column are not associated with a coded character set and therefore are never converted.
G The CCSID of a bit data column is X'FFFF'. In DB2 UDB for UWO, the
G CCSID for a bit data column is X'0000'.

If this clause is omitted, the CCSID of a SBCS, graphic, or mixed data column is the corresponding default CCSID at the current server.

CLOB(*integer [K | M | G]*)
For a character large object string of the specified maximum length in bytes. The maximum length must be in the range of 1 through 2 147 483 647. A CLOB column has a varying length. It cannot be referenced in certain contexts regardless of its maximum length. For details, see "Limitations on Use of Strings" on page 48.

G To create LOB columns in DB2 UDB for z/OS and OS/390, there are
G additional requirements. To create LOB columns greater than 1 gigabyte in
G DB2 UDB for UWO, there are additional requirements. See product
G documentation.

The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

integer
The maximum value for *integer* is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**
The maximum value for *integer* is 2 097 152. The maximum length is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If a value that evaluates to 2 gigabytes (2 147 483 648) is specified, then the value that is actually used is one byte less, that is 2 147 483 647.

GRAPHIC(*integer***)**
GRAPHIC
For a fixed-length graphic string of length *integer*. The integer can range from 1 to 127. See Table 39 on page 510 for more information.

VARGRAPHIC(*integer***)**
For a varying-length graphic string of maximum length *integer*. The integer can range from 1 to 16 336. See Table 39 on page 510 for more information.

For the restrictions that apply to the use of VARGRAPHIC strings longer than 127, see "Limitations on Use of Strings" on page 48.

DBCLOB(*integer [K | M | G]***)**
For a double-byte character large object string of the specified maximum length in double-byte characters. The maximum length must be in the range of 1 through 1 073 741 823. A DBCLOB column has a varying length. It cannot be referenced in certain contexts regardless of its maximum length. For details, see "Limitations on Use of Strings" on page 48.

G To create LOB columns in DB2 UDB for z/OS and OS/390, there are
G additional requirements. To create LOB columns greater than 1 gigabyte in
G DB2 UDB for UWO, there are additional requirements. See product
G documentation.

The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

integer
The maximum value for *integer* is 1 073 741 823. The maximum length of the string is *integer*.

integer **K**
The maximum value for *integer* is 1 048 576. The maximum length is 1024 times *integer*.

integer **M**
The maximum value for *integer* is 1024. The maximum length is 1 048 576 times *integer*.

integer **G**
The maximum value for *integer* is 1. The maximum length is 1 073 741 824 times *integer*.

If a value that evaluates to 2 gigabytes (1 073 741 824 double-byte characters) is specified, then the value that is actually used is one double-byte character less, that is 1 073 741 823.

BLOB(*integer [K | M | G]***)**
For a binary large object string of the specified maximum length in bytes. The maximum length must be in the range of 1 through 2 147 483 647. A

BLOB column has a varying length. It cannot be referenced in certain contexts regardless of its maximum length. For details, see “Limitations on Use of Strings” on page 48.

G G G G To create LOB columns in DB2 UDB for z/OS and OS/390, there are additional requirements. To create LOB columns greater than 1 gigabyte in DB2 UDB for UWO, there are additional requirements. See product documentation.

The maximum value that can be specified for *integer* depends on whether a units indicator is also specified as shown in the following list.

integer
: The maximum value for *integer* is 2 147 483 647. The maximum length of the string is *integer*.

integer **K**
: The maximum value for *integer* is 2 097 152. The maximum length is 1024 times *integer*.

integer **M**
: The maximum value for *integer* is 2048. The maximum length is 1 048 576 times *integer*.

integer **G**
: The maximum value for *integer* is 2. The maximum length is 1 073 741 824 times *integer*.

If a value that evaluates to 2 gigabytes (2 147 483 648) is specified, then the value that is actually used is one byte less, that is 2 147 483 647.

DATE
: For a date.

TIME
: For a time.

TIMESTAMP
: For a timestamp.

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 in the SQL path.

DEFAULT
: Specifies a default value for the column. This clause must not be specified more than once in the same *column-definition*.

Omission of NOT NULL and DEFAULT from a *column-definition* is an implicit specification of DEFAULT NULL.

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 58. 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 as the default for the column. The data type of the column or 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 the USER special register.

NULL
Specifies null as the default for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same *column-definition*.

cast-function-name
The name of the cast function that matches the name of the distinct type name of the data type for the column.

The schema name of the cast function, whether it is explicitly specified or implicitly resolved through function resolution, must be the same as the explicitly or implicitly specified schema name of the distinct type. This form of the DEFAULT value can only be used with columns that are defined as a 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.

USER
Specifies the value of the USER special register at the time of INSERT as the default as the default for the column. The source type of the distinct type of the column must be a CHAR or VARCHAR with a length attribute that is greater than or equal to the length attribute of the USER special register.

If the value specified is not valid, an error is returned.

NOT NULL
Prevents the column from containing null values. Omission of NOT NULL implies that the column can contain null values.

column-constraint
The *column-constraint* of a *column-definition* provides a shorthand method of defining a constraint composed of a single column. Thus, if a *column-constraint* is specified in the definition of column C, the effect is the same as if that constraint were specified as a *unique-constraint*, *referential-constraint*, or *check-constraint* in which C is the only identified column.

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not be the same as a constraint name 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.[58] Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

The NOT NULL clause must be specified with this clause. 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 column.

UNIQUE
Provides a shorthand method of defining a unique key composed of a single column.[58] Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE(C) clause is specified as a separate clause.

The NOT NULL clause must be specified with this clause. This clause cannot be specified more than once in a column definition and must not be specified if the PRIMARY KEY clause is specified in the column definition. The column must not be a LOB 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. The column must not be a LOB column. For more information, see "REFERENCES clause" on page 363.

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. For more information, see "CHECK clause" on page 365.

End of column-definition

unique-constraint

unique-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not be the same as a constraint name 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

58. In DB2 UDB for z/OS and OS/390, the table is marked as unavailable until all the required indexes are explicitly created, unless the CREATE TABLE statement is processed by the schema processor.

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, and that column must be defined as NOT NULL.

The same column must not be identified more than once. No column can be a LOB column. The number of identified columns must not exceed 16 and the sum of their byte count (see "Byte Counts" on page 366) must not exceed 255. See Table 41 on page 511 for more information.

A unique index on the identified columns is created during the execution of the CREATE TABLE statement and this index is designated as the primary index of the table.[58]

UNIQUE(*column-name***,...)**
Defines a unique key composed of the identified columns.[58] The UNIQUE clause can be specified more than once. The identified columns should not 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, and that column must be defined as NOT NULL.

The same column must not be identified more than once. No column can be a LOB column. The number of identified columns must not exceed 16 and the sum of their byte count (see "Byte Counts" on page 366) must not exceed 255. See Table 41 on page 511 for more information.

A unique index on the identified columns is created during the execution of the CREATE TABLE statement.[58]

End of unique-constraint

referential-constraint

referential-constraint

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not be the same as a constraint name 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. No column can be a LOB column. The number of identified columns

must not exceed 16 and the sum of their byte count (see "Byte Counts" on page 366) must not exceed 255. See Table 41 on page 511 for more information.

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 current server, but it must not identify a catalog 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
G constraints are allowed, but not recommended. In DB2 UDB for z/OS and
G OS/390, duplicate referential constraints are ignored with a warning.

Let T2 denote the identified parent table and let T1 denote the table being
G created. For DB2 UDB for z/OS and OS/390, T2 must not be the table
G being created except when the statement is processed by the schema
G processor.

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 other attributes.

If a foreign key column is a distinct type, the data type of the corresponding column of the parent key must have the same distinct type.

(*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. No column can be a LOB column. The number of identified columns must not exceed 16 and the sum of their byte count (see "Byte Counts" on page 366) must not exceed 255. See Table 41 on page 511 for more information.

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. 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 four possible actions:

- NO ACTION (default) [59]
- RESTRICT
- CASCADE
- SET NULL

59. In DB2 UDB for z/OS and OS/390, the default depends on the value of the CURRENT RULES special register when the CREATE TABLE statement is processed. If the value of the register is 'DB2', the default is RESTRICT. If the value is 'SQL', the default is NO ACTION.

SET NULL must not be specified unless some column of the foreign key allows null values.

G G G In DB2 UDB for UWO, a self-referencing table with a SET NULL or RESTRICT rule must not be a dependent in a referential constraint with a delete rule of CASCADE.

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 is returned and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the relationship would form a cycle and T2 is already delete-connected to T1, then the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. Let T3 denote a table identified in another FOREIGN KEY clause (if any) of the CREATE TABLE statement. The delete rules of the relationships involving T2 and T3 must be the same and must not be SET NULL if:

- T2 and T3 are the same table.
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3.
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2.
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3.

If *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

End of referential-constraint

check-constraint

check-constraint

CONSTRAINT *constraint-name*
Names the check constraint. A *constraint-name* must not be the same as a constraint name 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. [60]

The *check-condition* is a form of the *search-condition*, except:

- It can only refer to columns of the table whose data type is not a LOB data type or a distinct type based on a LOB data type.
- It can be up to 3800 bytes long, not including redundant blanks. See Table 41 on page 511 for more information.
- It must not contain any of the following:
  - subqueries
  - built-in functions
  - host variables
  - parameter markers
  - special registers
  - user-defined functions (except cast functions generated for distinct types)
  - CASE expressions

G In DB2 UDB for z/OS and OS/390, the *check-condition* is subject to additional
G restrictions. See the product reference for further information.

For more information about *search-condition*, see "Search Conditions" on page 126.

End of check-constraint

LIKE

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.

If a table is identified, then the implicit definition includes the following attributes of the columns of *table-name* (if applicable to the data type). If a view is identified, then the implicit definition includes the following attributes of each of the result columns of *view-name* (if applicable to the data type).

- column name
- data type, length, precision and scale
- CCSID
- nullability

For base tables, the default value attribute is also included in the table
G definition. For a view, if the column of the underlying base table has a default
G value, then the effect is product-specific.

60. In DB2 UDB for z/OS and OS/390, the value of the CURRENT RULES special register must be 'STD' to get this behavior.

The implicit definition does not include any other attributes of the identified table or view. For example, the new table does not have any primary key or foreign key.

End of LIKE

Notes

Owner Privileges: The owner of the table has all table provileges (see "GRANT (Table or View Privileges)" on page 418) with the ability to grant these privileges to others.

Creating Referential Constraints: The creation of referential constraints may invalidate access plans. The rules are product-specific.

Automatic Generation of Indexes: Whether an index name is generated and if so, the rules for generating the name of an index that is created during the execution of the CREATE TABLE statement are product-specific.

CCSIDs for Character and Graphic Columns: The CCSID of a SBCS, graphic, or mixed data column is the corresponding default CCSID at the current server.

Byte counts: The sum of the byte counts of the columns must not be greater than 32 677. See Table 41 on page 511 for more information.

The following table contains the byte counts of columns by data type for columns that do not allow null values. In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO, a column that allows null values has a byte count that is one more than shown in the list. In DB2 UDB for iSeries, if any column allows null values, one byte is required for every eight columns.

Data Type	Byte Count
SMALLINT	2
INTEGER	4
DECIMAL(*p*,*s*)	the integral part of (*p*/2) + 1
NUMERIC(*p*,*s*)	In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO, the integral part of (p/2)+1. In DB2 UDB for iSeries, p.
FLOAT (single precision)	4
FLOAT (double precision)	8
CHAR(*n*)	*n*
VARCHAR(*n*)	In DB2 UDB for z/OS and OS/390 and DB2 UDB for iSeries, n+2. In DB2 UDB for UWO, n+4.
CLOB(*n*)	Product-specific.
GRAPHIC(*n*)	2*n*
VARGRAPHIC(*n*)	In DB2 UDB for z/OS and OS/390 and DB2 UDB for iSeries, (n*2)+2. In DB2 UDB for UWO, (n*2)+4.
DBCLOB(*n*)	Product-specific.

	Data Type	Byte Count
G	BLOB(*n*)	Product-specific.
	DATE	4
	TIME	3
	TIMESTAMP	10
	distinct type	The byte count for the source data type.

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))
```

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

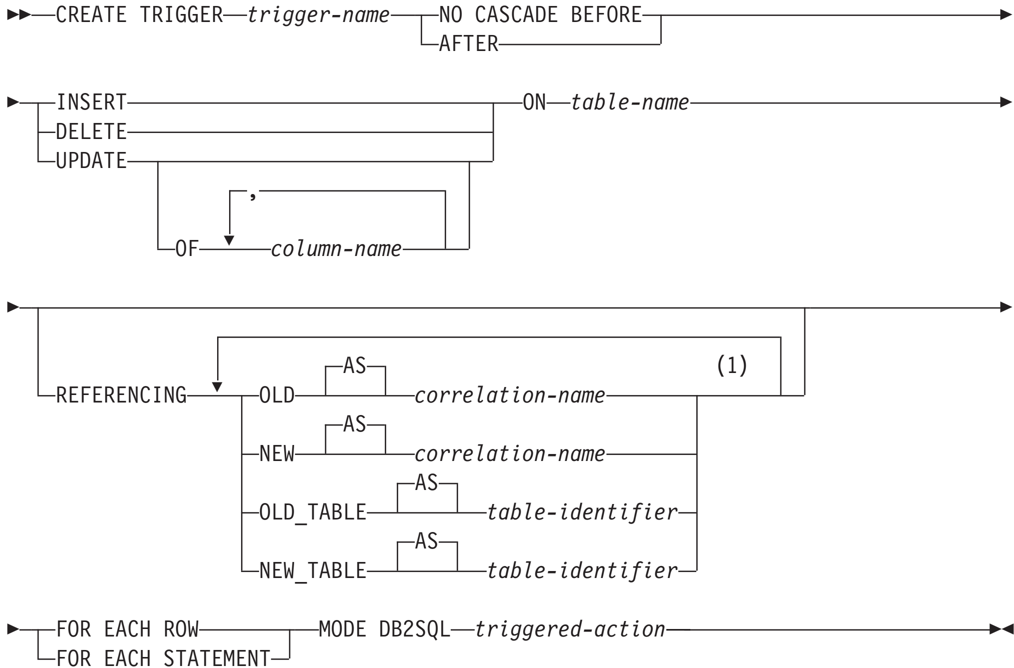
The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Administrative authority

The privileges held by the authorization ID of the statement must include at least one of the following:

- Each of the following:
  - The ALTER privilege on the table on which the trigger is defined,
  - The SELECT privilege on the table on which the trigger is defined,
  - The SELECT privilege on any table or view referenced in the *triggered-action search-condition*, and
  - The privileges required to execute each *triggered-SQL-statement*.
- Administrative authority.

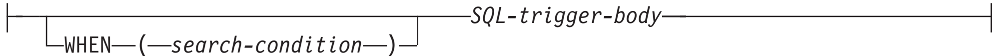
Syntax



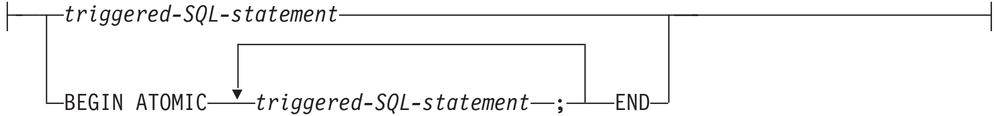
Notes:

1 The same clause must not be specified more than once.

triggered-action:



SQL-trigger-body:



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. If a qualified trigger name is specified, the *schema-name* must not be one of the system schemas (see "Schemas" on page 3).

NO CASCADE BEFORE
: Specifies that the trigger is a BEFORE trigger. The database manager executes the *triggered-action* before it applies any changes caused by its applicable 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 its applicable insert, delete, or update operation on the subject table.

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 or an alias.

REFERENCING
Specifies the correlation names for the transition variables and the table names for the transition tables. *Correlation-names* identify a specific row in the set of rows affected by the triggering SQL operation. *Table-identifiers* identify the complete set of affected rows.

Each row affected by the triggering SQL operation is available to the *triggered-action* by qualifying columns with *correlation-names* specified as follows:

OLD AS *correlation-name*
Specifies a correlation name that identifies the values in the row prior to the triggering SQL operation.

NEW 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 *table-identifiers* 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.

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-identifier*s 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 *correlation-name* and the NEW_TABLE *table-identifier* are valid only if the triggering event is either an insert operation or an update operation. For both operations, the NEW *correlation-name* captures the values of the columns in the inserted or updated row, and the NEW_TABLE *table-identifier* captures the values in the set of inserted or updated rows. For BEFORE triggers, the values of the updated rows include the changes from any SET statements in the *triggered-action* of BEFORE triggers.

The OLD and NEW *correlation-name* variables cannot be modified in an AFTER trigger.

The table below summarizes the allowable combinations of transition variables and transition tables.

<table>
<tr><th>Granularity</th><th>Activation Time</th><th>Triggering Operation</th><th>Transition Variables Allowed</th><th>Transition Tables Allowed</th></tr>
<tr><td rowspan="6">FOR EACH ROW</td><td rowspan="3">BEFORE</td><td>DELETE</td><td>OLD</td><td rowspan="3">NONE</td></tr>
<tr><td>INSERT</td><td>NEW</td></tr>
<tr><td>UPDATE</td><td>OLD, NEW</td></tr>
<tr><td rowspan="3">AFTER</td><td>DELETE</td><td>OLD</td><td>OLD_TABLE</td></tr>
<tr><td>INSERT</td><td>NEW</td><td>NEW_TABLE</td></tr>
<tr><td>UPDATE</td><td>OLD, NEW</td><td>OLD_TABLE, NEW_TABLE</td></tr>
<tr><td rowspan="6">FOR EACH STATEMENT</td><td rowspan="3">BEFORE</td><td>DELETE</td><td rowspan="3">NONE</td><td rowspan="3">NONE</td></tr>
<tr><td>INSERT</td></tr>
<tr><td>UPDATE</td></tr>
<tr><td rowspan="3">AFTER</td><td>DELETE</td><td rowspan="3">NONE</td><td>OLD_TABLE</td></tr>
<tr><td>INSERT</td><td>NEW_TABLE</td></tr>
<tr><td>UPDATE</td><td>OLD_TABLE, NEW_TABLE</td></tr>
</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 and 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.

MODE DB2SQL
Specifies the mode of the trigger. MODE DB2SQL triggers are activated after all of the row operations have occurred.

triggered-action
Specifies the action to be performed when 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.

WHEN (*search-condition***)**
Specifies a condition that evaluates to true, false, or unknown. The triggered SQL statements are executed only if the *search-condition* evaluates to true. If the WHEN clause is omitted, the associated SQL statements are always executed.

The *search-condition* for a BEFORE trigger must not include a subselect that references the subject table.

SQL-trigger-body
Specifies the SQL statements that are to be executed for the *triggered-action*.

triggered-SQL-statement
Specifies a single SQL statement that is to be executed for the *triggered-action*.

BEGIN ATOMIC *triggered-SQL-statement; ...* **END**
Specifies a list of SQL statements that are to be executed for the *triggered-action*. The statements are executed in the order in which they are specified.

Only certain SQL statements can be specified in the *SQL-trigger-body*. The following table shows the list of allowable SQL statements, which differs depending on whether the trigger is defined as BEFORE or AFTER. An 'X' in the table indicates that the statement is valid.

SQL statement	BEFORE	AFTER
"fullselect" on page 248	X	X
"SET transition-variable" on page 464	X	

“SIGNAL Statement” on page 504	X	X
“VALUES” on page 472	X	X
“INSERT” on page 423		X
Searched “DELETE” on page 386		X
Searched “UPDATE” on page 466		X

All tables, views, aliases, user-defined 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.

A fullselect specified in a BEFORE trigger must not refer to the subject table of the trigger.

Notes

Owner privileges: There are no specific privileges on a trigger. For more information on ownership of an object, see “Authorization, Privileges and Object Ownership” on page 27.

Execution Authorization: The user executing the triggering SQL operation does not need authority to execute a *triggered-SQL-statement*. A *triggered-SQL-statement* will execute using the authority of the *owner* of the trigger.

Activating a trigger: Only insert, delete, or update operations can activate a trigger. The activation of a trigger 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 number of levels of nested trigger cascading is limited to 16. For more information see Appendix A, “SQL Limits” on page 509.

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 table. For more information see Appendix A, “SQL Limits” on page 509.

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 subject table is referenced in the *triggered-action*, 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 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 that include those references to the object will be rebuilt when the trigger is activated. 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 activated. If the appropriate privilege does not exist at that time, the corresponding insert, update or delete operation on the subject table will fail.

G DB2 UDB for UWO effectively drops the trigger when a dependent object is
G dropped or a required privilege is revoked.

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 typically 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 the statements in the *SQL-trigger-body* run under the
G isolation level of the triggering SQL operation. In DB2 UDB for z/OS and OS/390
G the SQL statements in the *SQL-trigger-body* run under the isolation level used at the
G time the trigger was created.

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 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

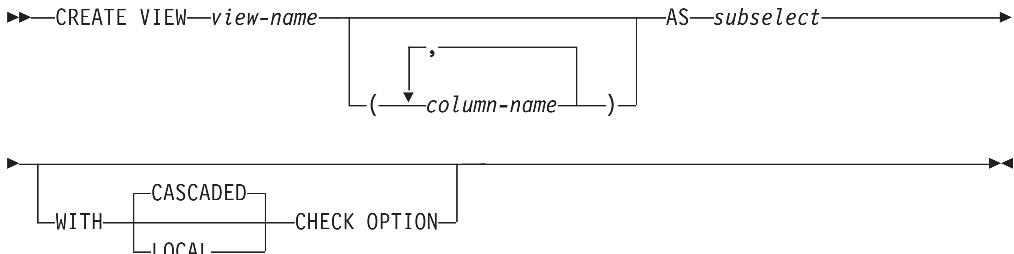
The privileges held by the authorization ID of the statement must include at least one of the following:

- The privilege to create in the schema
- Administrative authority.

The privilege held by the authorization ID of the statement must include at least one of the following:

- For each table or view identified in the *subselect*:
  - The SELECT privilege on the table or view
  - Ownership of the table or view.
- Administrative authority

Syntax



Description

view-name
: Names the view. The name, including the implicit or explicit qualifier, must not identify an alias, index, table or view that already exists at the current server.

(*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 *subselect*. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the view inherit the names of the columns of the result table of the *subselect*.

  A list of column names must be specified if the result table of the *subselect* has duplicate column names or an unnamed column. For more information about unnamed columns, see “Names of result columns” on page 237.

AS *subselect*
: Defines the view. At any time, the view consists of the rows that would result if the *subselect* were executed.

subselect must not reference host variables.

The maximum number of columns allowed in a view is 750. The maximum number of base tables allowed in a view is 32. See Table 41 on page 511 for more information.

For an explanation of *subselect*, see "subselect" on page 234.

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.

The CHECK OPTION clause must not be specified if the view is read-only or includes a subquery. If the CHECK OPTION clause is specified for an updatable view that does not allow inserts, then it applies to updates only.

The CHECK OPTION clause must not be specified if the view references a non-deterministic function.

If the CHECK OPTION clause is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes the CHECK OPTION clause. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of the view.

The difference between the two forms of the CHECK OPTION clause, CASCADED and LOCAL, is meaningful only when a view is dependent on another view. The default is CASCADED. The view upon which another view is directly or indirectly defined is an *underlying* view.

CASCADED

The WITH CASCADED CHECK OPTION on a view V is inherited by any updatable view that is directly or indirectly dependent on V. Thus, if V is an underlying view for an updatable view, the CHECK OPTION clause on V also applies to that view, even if the CHECK OPTION clause is not specified on that view. The search conditions of V and each view which is an underlying view for V are ANDed together to form a search condition that is applied for an insert or update of V or of any view dependent on V.

Consider the following updatable views which shows the impact of the WITH CASCADED CHECK OPTION:

```
CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10

CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CASCADED CHECK OPTION

CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100
```

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.
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.

SQL statement	Description of result
INSERT INTO V3 VALUES(200)	Succeeds even though it does not conform to the definition of V3 (V3 does not have the view CHECK OPTION clause specified); it does conform to the definition of V2 (which does have the view CHECK OPTION clause specified).

LOCAL

The WITH LOCAL CHECK OPTION on a view V means the search condition of V is applied as a constraint for an insert or update of V or of any view that is dependent on V. 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 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 updatable 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
```

This example shows V1 as *an underlying view for* V2 and V2 *dependent on* V1.

The following table describes which search conditions are checked during an INSERT or UPDATE operation:

Table 35. Views whose search conditions are checked during INSERT and UPDATE

View used in INSERT or UPDATE operation	x = LOCAL y = LOCAL	x = CASCADED y = CASCADED	x = LOCAL y = CASCADED	x = CASCADED y = LOCAL
V1	None	None	None	None
V2	V2	V2, V1	V2	V2, V1
V3	V2	V2, V1	V2	V2, V1
V4	V4, V2	V4, V3, V2, V1	V4, V3, V2, V1	V4, V2, V1
V5	V4, V2	V4, V3, V2, V1	V4, V3, V2, V1	V4, V2, V1

Notes

Owner Privileges: The owner always acquires the SELECT privilege on the view. The SELECT privilege can be granted to others only if the owner has the authority to grant the SELECT privilege on every table or view identified in the first FROM clause of the *subselect*.

The owner can 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 *subselect*. The privileges can be granted only if the privileges from which they are derived also can be granted. The owner only acquires these privileges if the privileges from which they are derived exist at the time the view is created. For more information on ownership of objects see “Authorization, Privileges and Object Ownership” on page 27.

Deletable views: A view is *deletable* if all of the following are true:

- the FROM clause of the outer *subselect* identifies only one base table, deletable view, or deletable nested table expression (that is, a nested table expression whose *subselect*, if used to create a view, would create a deletable view) that is not a catalog table or view
- the outer *subselect* does not include a GROUP BY clause or HAVING clause
- the outer *subselect* does not include column functions in the select list
- the select-clause of the outer *subselect* does not include DISTINCT
- no base table (or underlying base table of a view) in a subquery contained in the *subselect* is the same as the base table (or underlying base table of a view) in the outer *subselect*

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 updatable.

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 all columns of the view are updatable.

If a view contains two updatable columns that refer to the same column in the underlying table, the view is not insertable.

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.

Examples

Example 1: Create a view named MA_PROJ upon the PROJECT table that contains only those rows with a project number (PROJNO) starting with the letters ‘MA’.

```
CREATE VIEW MA_PROJ
  AS SELECT * FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 2: Create a view as in example 1, but select only the columns for project number (PROJNO), project name (PROJNAME) and employee in charge of the project (RESPEMP).

```
CREATE VIEW MA_PROJ
  AS SELECT PROJNO, PROJNAME, RESPEMP FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 3: Create a view as in example 2, but, in the view, call the column for the employee in charge of the project IN_CHARGE.

```
CREATE VIEW MA_PROJ (PROJNO, PROJNAME, IN_CHARGE)
  AS SELECT PROJNO, PROJNAME, RESPEMP FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Note: Even though only one of the column names is being changed, the names of all three columns in the view must be listed in the parentheses that follow MA_PROJ.

Example 4: Create a view named PRJ_LEADER that contains the first four columns (PROJNO, PROJNAME, DEPTNO, RESPEMP) from the PROJECT table together with the last name (LASTNAME) of the person who is responsible for the project (RESPEMP). Obtain the name from the EMPLOYEE table by matching EMPNO in EMPLOYEE to RESPEMP in PROJECT.

```
CREATE VIEW PRJ_LEADER
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME
       FROM PROJECT, EMPLOYEE
       WHERE RESPEMP = EMPNO
```

Example 5: Create a view as in example 4, but in addition to the columns PROJNO, PROJNAME, DEPTNO, RESPEMP and LASTNAME, show the total pay (SALARY + BONUS + COMM) of the employee who is responsible. Also select only those projects with mean staffing (PRSTAFF) greater than one.

```
CREATE VIEW PRJ_LEADER (PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, TOTAL_PAY)
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, SALARY+BONUS+COMM
       FROM PROJECT, EMPLOYEE
       WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

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
  - Ownership of the table or view
- Administrative authority.

The SELECT statement of the cursor is one of the following:

- The prepared *select-statement* identified by the *statement-name*
- The specified *select-statement*.

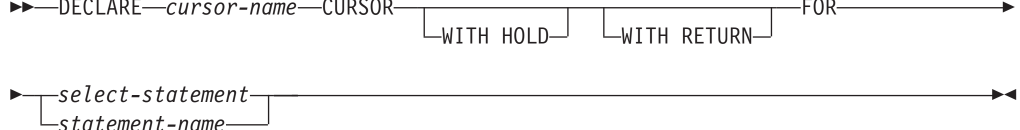
If *statement-name* is specified:

- The authorization ID of the statement is the run-time authorization ID.
- The authorization check is performed when the *select-statement* is prepared.
- The cursor cannot be opened unless the *select-statement* is successfully prepared.

If *select-statement* is specified:

- The authorization ID of the statement is the authorization ID specified during program preparation.
- In REXX, the authorization ID of the statement is the run-time authorization ID.
- Depending on the product environment or options, the authorization check is performed either during program preparation, or when the cursor is opened. See the product references for further information.

Syntax



Description

cursor-name
: Names the cursor. The name must not be the same as the name of another cursor declared in the source program.

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 by a commit only if the connection 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. 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 cursor 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.

To return a result set from an external procedure, the cursor must be declared using the WITH RETURN clause. All other cursors must be closed using the CLOSE statement. For Java external procedures, all cursors are implicitly declared WITH RETURN.

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 250 for more information.

The *select-statement* must not include parameter markers (except for REXX), but can include references to host variables. In host languages, other than REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. In REXX, parameter markers must be used in place of host variables and the statement must be prepared.

statement-name

Specifies the prepared *select-statement* that specifies the result table of the cursor whenever the cursor is opened. The *statement-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of the source program. See “PREPARE” on page 433 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: [61]

61. In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO, a program preparation option must be used if the UPDATE clause is not specified in the *select-statement* of the cursor and for DB2 UDB for UWO if the cursor is not statically defined. For DB2 UDB for z/OS and OS/390, use the precompiler option STDSQL(YES) or NOFOR. For DB2 UDB for UWO, use the program preparation option LANGLEVEL SQL92E.

- each FROM clause of the outer fullselect identifies only one base table or deletable view (cannot identify a nested table expression)
- 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 UNION or UNION ALL
- the select-clause of the outer fullselect does not include DISTINCT
- the *select-statement* does not include an ORDER BY clause
- the *select-statement* does not include a 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
- no base table (or underlying base table of a view) in a subquery contained in the fullselect is the same as the base table (or underlying base table of a view) in the outer fullselect
- if it is executed with isolation level UR, then the UPDATE clause must be specified

A column in the select list of the outer fullselect associated with a cursor is *updatable* if all of the following are true:[61]
- 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 UPDATE is specified without a list of column names, only the updatable columns in the SELECT clause of the *subselect* can be updated. If the UPDATE clause of the *select-statement* of the cursor is specified with a list of column names, only the columns specified in the list of column names can be updated.

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 in 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. Cursors that specify WITH RETURN in a procedure and are left open are returned as result sets.

Although the scope of a cursor is the program in which it is declared, each package 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 to be returned 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. If the SELECT statement of a cursor contains CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, all references to these special registers will yield the same respective datetime value on each FETCH. The value is determined when the cursor is opened. Multiple cursors using the same SELECT statement can be opened concurrently. They are each considered independent activities.

Blocking of data: For more efficient processing of data, the database manager may block data for read-only cursors. If a cursor is not going to be used in a Positioned UPDATE or Positioned 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.

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 EMPPROJACT 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 EMPPROJACT \
                 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.     */
    .
    .
    .
    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;
    .
    .
    .
  }
```

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 can 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:

- The DELETE privilege for the table or view
- Ownership of the table [62]
- Administrative authority.

If *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 for the table or view [63]
- Ownership of 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:

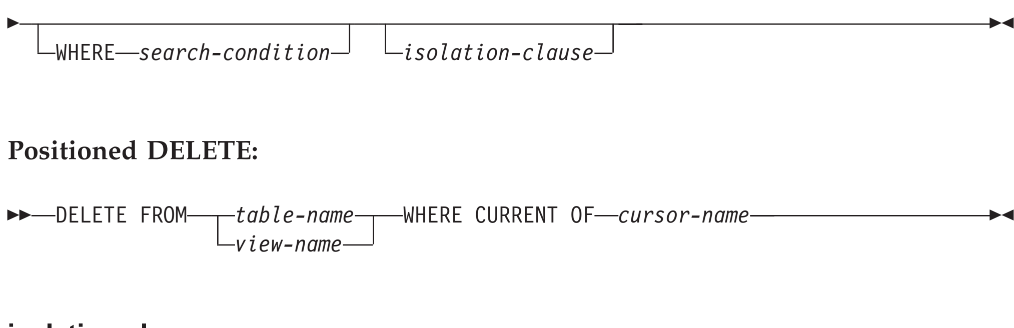
- For every table or view identified in the subquery:
  - The SELECT privilege on the table or view, or
  - Ownership of the table or view.
- Administrative authority.

Syntax

Searched DELETE:

62. The DELETE privilege on a view is only inherent in administrative authority. Ownership of a view does not necessarily include the DELETE privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

63. In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, the authorization ID of the statement only requires the DELETE privilege for the table or view. To require the SELECT privilege, a standards option must be in effect. For DB2 UDB for z/OS and OS/390 use the precompiler option SQLRULES(STD) or set the CURRENT RULES special register to 'STD'. For DB2 UDB for UWO, use the program preparation option LANGLEVEL SQL92E.



isolation-clause:

```
├──WITH──┬─RR─┬────────────────────────────────────────────────────────┤
         ├─RS─┤
         └─CS─┘
```

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 376.

correlation-name
: Can be used within the *search-condition* to designate the table or view. For an explanation of *correlation-name*, see “Correlation Names” on page 79.

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
: Specifies a search condition, as described in “Search Conditions” on page 126. 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 *search-condition* is true.

If *search-condition* contains a subquery, the subquery can be thought of as being executed each time the *search condition* is applied to a row, and the results used in applying the *search condition*. In actuality, a subquery with no correlated references may be executed only once, whereas a subquery with a correlated reference may have to be executed once for each row.

Let T2 denote the object table of a DELETE statement and let T1 denote a table that is referenced in the FROM clause of a subquery of that statement. T1 must not be a table that can be affected by the DELETE on T2. Thus, the following rules apply:

- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL.
- T1 must not be a dependent of T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

CURRENT OF *cursor-name*
Identifies the cursor to be used in the delete operation. The *cursor-name* must identify a declared cursor as explained in the *Notes* section of "DECLARE CURSOR" on page 381.

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 381.

When the DELETE statement is executed, the cursor must be open and positioned on a row and that row is 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.

G In DB2 UDB for z/OS and OS/390, if the DELETE statement is embedded
G in a program, the DECLARE CURSOR statement must include a
G *select-statement* rather than a *statement-name*.

isolation-clause
Specifies the isolation level used by the statement.

WITH

Introduces the isolation level, which may be one of:
- RR Repeatable read
- RS Read stability
- CS Cursor stability

If *isolation-clause* is not specified, the default isolation is used. For more information on the default isolation, see "Isolation Level" on page 13.

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 return error conditions based on the deleted values.

Referential Integrity: If the identified table or the base table of the identified view is a parent, the rows selected must not have any dependents in a relationship with a delete rule of RESTRICT or NO ACTION, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT or NO ACTION (SQLSTATE 23504).

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.
- 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.

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. If deleting a row in the parent table would cause a column in a dependent table to be set to null and that null value would cause the search condition of a check constraint to evaluate to false, the row is not deleted (SQLSTATE 23511).

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.

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.

The locks can prevent other application processes from performing operations on the table.

Position of cursor: If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of their result table. Let C be a cursor that is positioned before row R (as a result of an OPEN, a DELETE through C, a DELETE through some other cursor, or a Searched DELETE). In the presence of INSERT, UPDATE, and DELETE operations that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R’, where R’ is a new row that is now the next row of the result table.

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 C, “SQL Communication Area (SQLCA)” on page 525.

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 433.

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 433 for the authorization required to create a prepared statement.

Syntax

```
▸▸──DESCRIBE──statement-name──INTO──descriptor-name──────────────────────────▸◂
```

Description

statement-name
: Identifies the prepared statement. When the DESCRIBE statement is executed, the name must identify a prepared statement at the current server.

INTO *descriptor-name*
: Identifies an SQL descriptor area (SQLDA). For more information, see Appendix D, "SQL Descriptor Area (SQLDA)" on page 529. 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 entries required, see "Determining How Many Occurrences of SQLVAR Entries are Needed" on page 531.

The rules for REXX are different. For more information, see Appendix K, "Coding SQL Statements in REXX Applications" on page 643.

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 SQLVAR entries for every select list item (or, column of the result table), the seventh byte is set to '2'. This technique is used in order to accommodate LOB or distinct type result columns.
- 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 entries..

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 entries so that the first occurrence of an SQLVAR entry contains a description of the first column of the result table, the second occurrence of SQLVAR entry contains a description of the second column of the result table, and so on. For information on the values assigned to SQLVAR entries, see “Field Descriptions in an Occurrence of SQLVAR” on page 532.

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 and COBOL, 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 Occurrences of SQLVAR Entries are Needed” on page 531 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 * SQLD occurrences and set SQLN in the new SQLDA to 2 * 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.

Examples

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 entrires 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  initialsqlda;
struct sqlda *sqldaPtr;

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 SQLDABC to length of SQLDA */
EXEC SQL  DESCRIBE STMT1_NAME INTO :initialsqlda;

if (initialsqlda.sqld > 0)  /* statement is a select-statement */
  {

   ... /* Code to allocate correct size SQLDA (sets sqldaPtr)  */

   if (strcmp(SQLSTATE,"01005") == 0)
     {
      sqldaPtr->sqln = 2*initialsqlda.sqld;
      SETSQLDOUBLED(sqldaPtr, SQLDOUBLED);
     }
   else
     {
      sqldaPtr->sqln = initialsqlda.sqld;
      SETSQLDOUBLED(sqldaPtr, SQLSINGLED);
     }
   EXEC SQL  DESCRIBE STMT1_NAME  INTO :*sqldaPtr;
```

```
    ... /* code to prepare for the use of the SQLDA             */
    EXEC SQL  OPEN DYN_CURSOR;

    ... /* loop to fetch rows from result table                 */
    EXEC SQL  FETCH DYN_CURSOR  USING DESCRIPTOR :*sqldaPtr;

    ...
   }
...
```

DROP

The DROP statement drops an object. Objects that are directly or indirectly dependent on that object may also be dropped.

Invocation

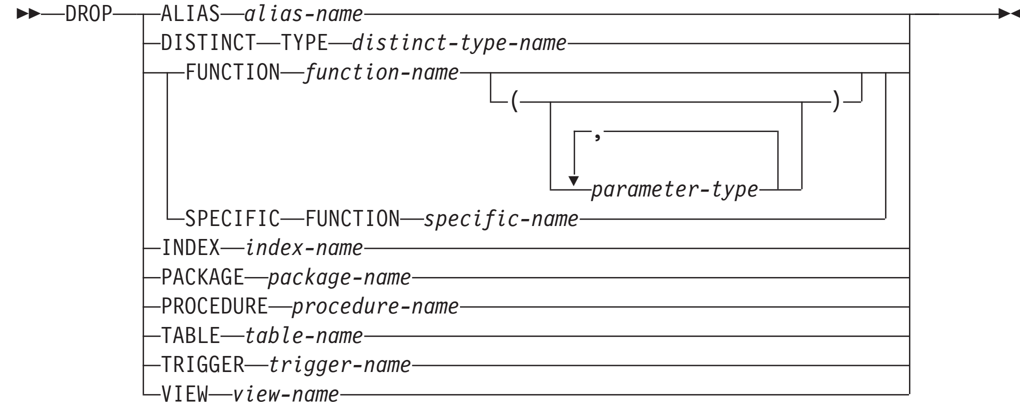
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the object
- Administrative authority.

Syntax



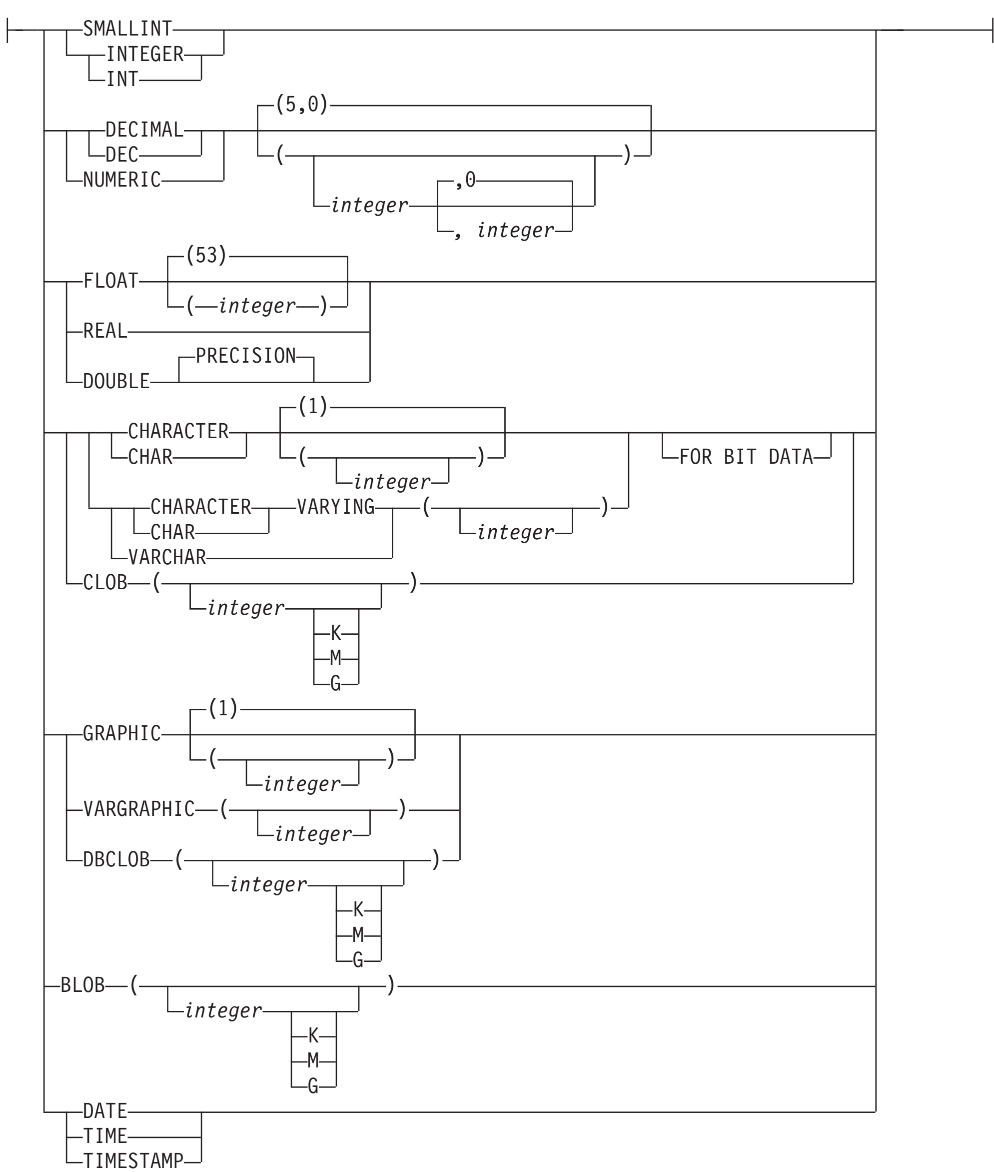
parameter-type:

```
|--data-type--+------------+--------------------------------------|
              '-AS LOCATOR-'
```

data-type:

```
|--+-built-in-type------+----------------------------------------|
   '-distinct-type-name-'
```

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 dropped from the schema. Dropping an alias has no
G effect on any constraint that was defined using the alias. The effect on any
G tables, views, routines, or triggers that reference the alias is product-specific.

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 dropped from the schema. All privileges on the distinct
G type are also dropped. The effect on any tables, views, routines, or triggers that
G reference the type is product-specific.

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.

The specified function is dropped from the schema. All privileges on the
G user-defined function are also dropped. The effect on any routines, triggers, or
G views that reference the function is product-specific.

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.

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, but it must not identify:

- A primary index.
- A unique index used to enforce a UNIQUE constraint.
- An index on a catalog table.

The specified index is dropped from the schema. See the product references for additional restrictions on dropping indexes.

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.

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
G procedure are also dropped. The effect on any routines that reference the
G procedure is product-specific.

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 it must not identify a catalog table.

The specified table is dropped from the schema. All privileges, constraints, indexes, triggers, and views on the table are also dropped. Any referential
G constraints in which the table is the parent are dropped. The effect on any
G routines or triggers that reference the table is product-specific.

Any aliases that reference the specified table are not dropped.

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.

VIEW *view-name*
Identifies the view that is to be dropped. The *view-name* must identify a view that exists at the current server.

The specified view is dropped from the schema. Any view that is directly or indirectly dependent on that view is also dropped. Whenever a view is

G G dropped, all privileges on that view are also dropped. The effect on any routines, or triggers that reference the view is product-specific.

Any aliases that reference the specified view are not dropped.

Notes

Drop effects: Whenever an object is dropped, its description is dropped from the catalog and any access plans that reference the object are invalidated. For more information, see "Packages and Access Plans" on page 9.

G G **Drop restriction:** In DB2 UDB for z/OS and OS/390, after an object is dropped, a commit must be performed before recreating the object with the same name.

Examples

Example 1: Drop the table named MY_IN_TRAY.

```
DROP TABLE MY_IN_TRAY
```

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.

```
DROP DISTINCT TYPE DOCUMENT
```

Example 5: Assume that ATOMIC_WEIGHT is the only function with that name in schema CHEM. Drop ATOMIC_WEIGHT.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT
```

Example 6: Drop the function named CENTER, using the function signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER (INTEGER, DOUBLE)
```

Example 7: Drop CENTER, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION JOHNSON.FOCUS97
```

Example 8: Assume that procedure OSMOSIS is in schema BIOLOGY. Drop OSMOSIS.

```
DROP PROCEDURE BIOLOGY.OSMOSIS
```

Example 9: Assume that trigger BONUS exists in the default 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 Java or REXX.

Authorization

None required.

Syntax

```
►►──END DECLARE SECTION──────────────────────────────────────────────►◄
```

Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear in accordance with the rules of the host language. It is used to indicate the end of an SQL declare section. An SQL declare section starts with a BEGIN DECLARE SECTION statement described in “BEGIN DECLARE SECTION” on page 281.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

Examples

See “BEGIN DECLARE SECTION” on page 281 for examples that use the END DECLARE SECTION statement.

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

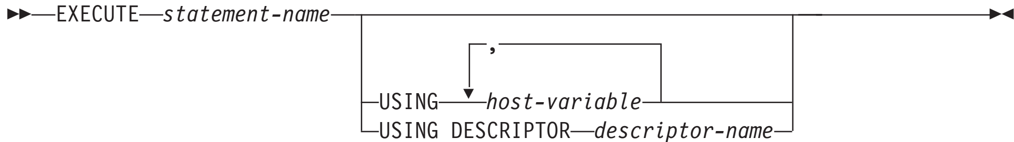
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

See “PREPARE” on page 433 for the authorization required to create a prepared statement.

Syntax



Description

statement-name
Identifies the prepared statement to be executed. When the EXECUTE statement is executed, the name must identify a prepared statement at the current server. 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 433.) 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 or 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 (Note that the rules for REXX are different. For more information, see Appendix K, “Coding SQL Statements in REXX Applications” on page 643):

- SQLN to indicate the number of SQLVAR entries 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 entries to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all occurrences of SQLVAR entries. If an SQLVAR entry includes a LOB or distinct type based on a LOB, there must be additional SQLVAR entries for each parameter. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR entries, see Appendix D, “SQL Descriptor Area (SQLDA)” on page 529.

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.

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 36 on page 435.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P using storage assignment rules as described in “Assignments and Comparisons” on page 58. Thus:

- V must be compatible with the target.
- If V is a string, its length must not be greater than the length attribute of the target.
- If V is a number, the whole part of the number must not be truncated.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, V must not be null.

When the prepared statement is executed, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6) and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.

Examples

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) BINARY.
  77 TIM             PIC S9(3)V9(2).
  01 HOLDER.
     49  HOLDER-LENGTH    PIC S9(4) BINARY.
     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 statement 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 423 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

The authorization ID is the runtime authorization ID.

Syntax

```
►►──EXECUTE IMMEDIATE──host-variable──────────────────────────────────────►◄
```

Description

host-variable

Identifies a host variable that must be described in accordance with the rules for declaring character-string host variables. The host variable must not have a CLOB data type, and an indicator variable must not be specified.

In COBOL it must be a varying-length string variable. In C, it must be the VARCHAR structured form of a string variable rather than the NUL-terminated form.

The value of the identified host variable is called the *statement string*.

The statement string must be one of the following SQL statements:[64]

ALTER	DROP	REVOKE
COMMENT	GRANT	ROLLBACK
COMMIT	INSERT	SET PATH
CREATE	LOCK TABLE	UPDATE
DELETE	RENAME	

The statement string must not:

64. A select-statement is not allowed. To dynamically process a select-statement, use the PREPARE, DECLARE CURSOR, and OPEN statements.

- Begin with EXEC SQL.
- End with END-EXEC or a semicolon.
- Include references to host variables.
- Include parameter markers.
- Include comments.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed and the error is returned. If the SQL statement is valid, but an error occurs during its execution, that error is returned.

Notes

Performance considerations: If the same SQL statement is to be executed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Examples

Use C to execute the SQL statement in the host variable Qstring.

```
EXEC SQL INCLUDE SQLCA;
void main ()
  {

    EXEC SQL BEGIN DECLARE SECTION;

    char Qstring[100] =
      "INSERT INTO WORK_TABLE SELECT * FROM EMPPROJACT WHERE ACTNO >= 100";

    EXEC SQL END DECLARE SECTION;

  .
  .
  .
    EXEC SQL EXECUTE IMMEDIATE :Qstring;

    return;
  }
```

FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables.

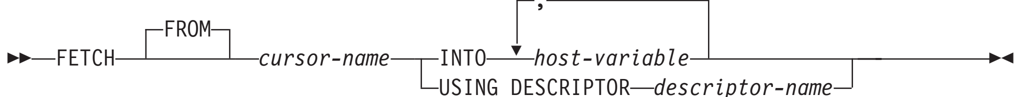
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See "DECLARE CURSOR" on page 381 for an explanation of the authorization required to use a cursor.

Syntax



Description

cursor-name
Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify either a declared cursor as explained in "DECLARE CURSOR" on page 381 or when used in Java, an instance of an SQLJ iterator. When the FETCH statement is executed, the cursor must be in the open state.

INTO *host-variable***,...**
Identifies one or more host structures or variables that must be described in accordance with the rules for declaring host structures and variables. In the operational form of INTO, a reference to a structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so on.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more output host variables.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA (note that the rules for REXX are different, for more information see Appendix K, "Coding SQL Statements in REXX Applications" on page 643):

- 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 + SQLN*(N), where N 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 529.

Notes

Cursor Position: An open cursor has three possible positions:
- Before a row
- On a row
- After the last row.

If the cursor is currently positioned on or after the last row of the result table:
- SQLSTATE is set to '02000'.
- The cursor is positioned after the last row.
- Values are not assigned to host variables.

If the cursor is currently positioned before a row, the cursor is positioned on that row, and the values of that row are assigned to host variables as specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, the cursor is positioned on the next row and values of that row are assigned to host variables as specified by INTO or USING.

If a cursor is on a row, that row is called the current row of the cursor. A cursor referenced in an UPDATE or DELETE statement must be positioned on a row. A cursor can only be on a row as a result of a FETCH statement.

It is possible for an error to occur that makes the state of the cursor unpredictable.

Host Variable Assignment: The *n*th variable identified by the INTO clause or described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of each variable must be compatible with its corresponding column.

Each assignment to a variable is made according to the Retrieval Assignment rules described in “Assignments and Comparisons” on page 58. 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 values in the host variables are unpredictable.

Result Column Evaluation Considerations: If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (such as 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 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 occurs.[65]

If the specified host variable is not large enough to contain the result, a warning is returned (SQLSTATE 01004) and 'W' is assigned to SQLWARN1 in the SQLCA. The actual length of the result is returned in the indicator variable associated with the host-variable, if an indicator variable is provided. If a CLOB, DBCLOB or BLOB value is truncated, the length may not be returned in the indicator variable.

It is possible that a warning may not be returned on a FETCH. This occurs as a result of optimizations such as the use of system temporary tables or blocking. It is also possible that the returned warning applies to a previously fetched row.

When a datetime value is returned, the length of the variable must be large enough to store the complete value. Otherwise, depending on how much of the value would have to be truncated, a warning or an error is returned. See "Datetime Assignments" on page 62 for details.

Examples

Example 1: In this C example, the FETCH statement fetches the results of the SELECT statement into the program variables dnum, dname, and mnum. When no more rows remain to be fetched, the not found condition is returned.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT
  WHERE ADMRDEPT = 'A00';
EXEC SQL OPEN C1;
while (SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
}
EXEC SQL CLOSE C1;
```

Example 2: This FETCH statement uses an SQLDA.

```
FETCH CURS USING DESCRIPTOR :sqlda3
```

65. In DB2 UDB for UWO, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

FREE LOCATOR

The FREE LOCATOR statement removes the association between a locator variable and its value.

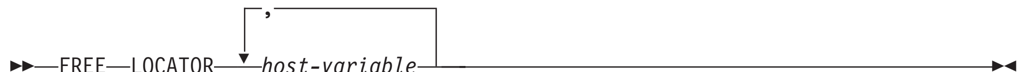
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java.

Authorization

None required.

Syntax



Description

host-variable, **...**
: Identifies one or more locator variables that must be declared in accordance with the rules for declaring locator variables. The locator variable type must be a binary large object locator, a character large object locator, or a double-byte character large object locator.

The *host-variable* must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a FETCH, SELECT INTO, assignment statement, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned.

If more than one locator is specified and an error is returned on one of the locators, it is possible that some locators have been freed and others have not been freed.

Examples

Assume that the employee table contains columns RESUME, HISTORY, and PICTURE and that the 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.

G In DB2 UDB for UWO, this statement is not supported. Instead, PUBLIC implicitly
G has the USAGE privilege on all distinct types.

Authorization

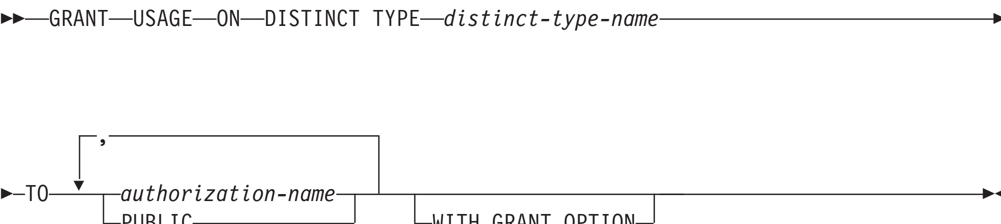
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the distinct type
- The USAGE privilege on the distinct type with the WITH GRANT OPTION
- 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

USAGE
Grants the privilege to use the distinct type in tables, functions, procedures, or CAST expressions.

ON DISTINCT TYPE *distinct-type-name*
Identifies the distinct type 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 privilege is granted.

authorization-name,**...**
Lists one or more authorization IDs.[66]

66. In DB2 UDB for z/OS and OS/390, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, Privileges and Object Ownership" on page 27.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the USAGE privilege to other users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the USAGE privilege to others unless they have received that authority from some other source.

Notes

Cast function implications: The GRANT (Distinct Type Privileges) statement does not grant a user the privilege to execute the cast functions that are associated with the distinct type. The GRANT (Function or Procedure Privileges) statement must be used to grant the EXECUTE privilege to the cast functions associated with the distinct type.

Examples

Grant the USAGE privilege on distinct type SHOE_SIZE to user JONES.

```
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.

In DB2 UDB for UWO, this statement is not supported. Instead, PUBLIC implicitly has the EXECUTE privilege on all functions and procedures.

Authorization

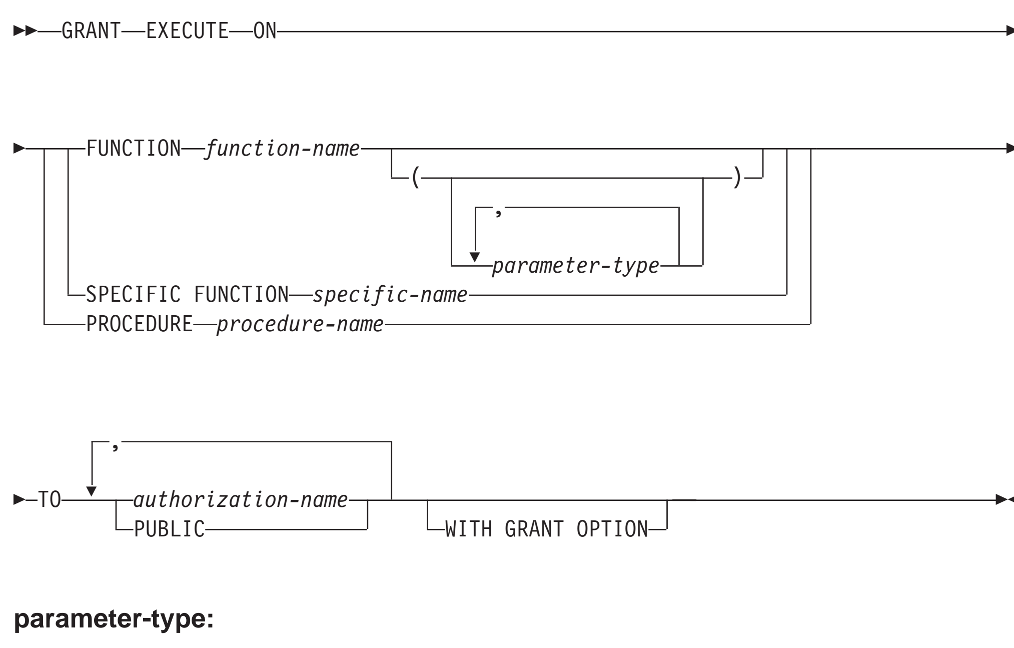
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the function or procedure
- The EXECUTE privilege on the function or procedure with the WITH GRANT OPTION
- 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.

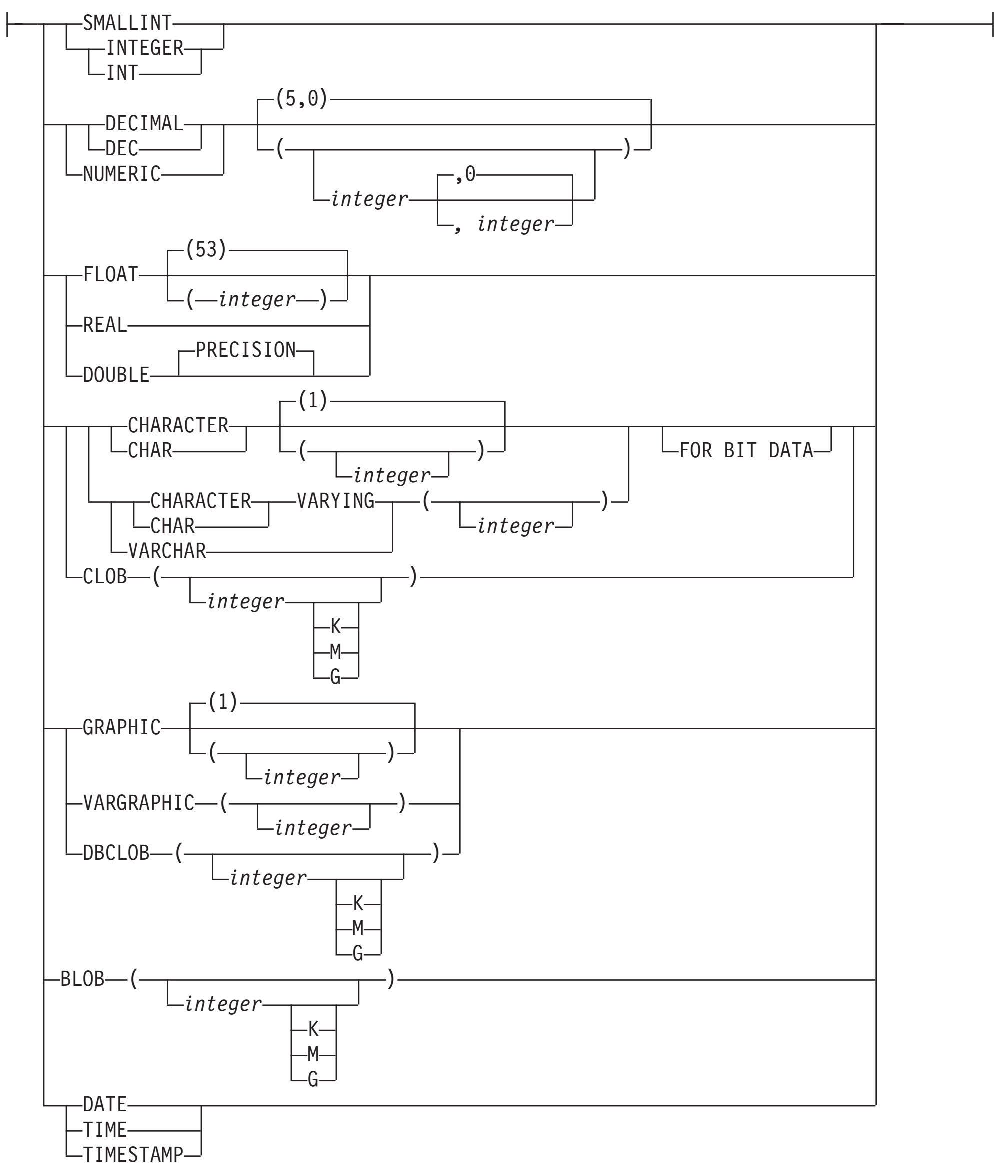
Syntax



data-type:

```
├──┬─built-in-type──────┬──────────────────────────────────┤
   └─distinct-type-name─┘
```

built-in-type:



Description

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 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 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.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure on which the privilege is granted. The *procedure-name* must identify a procedure that exists at the current server.

TO

Indicates to whom the privilege is granted.

authorization-name,**...**

Lists one or more authorization IDs.[67]

PUBLIC

Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, Privileges and Object Ownership" on page 27.

WITH GRANT OPTION

Allows the specified *authorization-name*s to grant the EXECUTE privilege to others users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the EXECUTE privilege to others unless they have received that authority from some other source.

Notes

Built-in Functions: Privileges cannot be granted on built-in functions.

Examples

Grant the EXECUTE privilege on procedure PROCA to PUBLIC.

```
GRANT EXECUTE
  ON PROCEDURE PROCA
  TO PUBLIC
```

67. In DB2 UDB for z/OS and OS/390, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

GRANT (Package Privileges)

This form of the GRANT statement grants the privilege to execute statements in 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

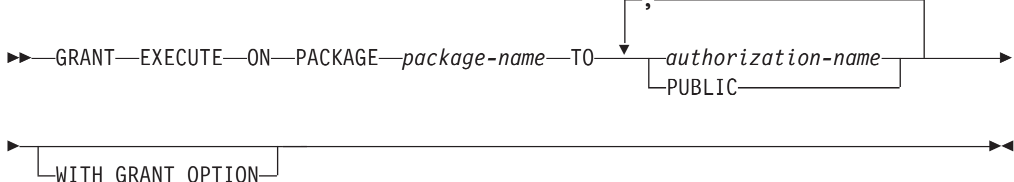
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the package
- The EXECUTE privilege on the package with the WITH GRANT OPTION
- 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

EXECUTE
Grants the privilege to execute SQL statements in a package.

ON PACKAGE *package-name*
Identifies the package on which the EXECUTE privilege is granted. The *package-name* must identify a package that exists at the current server.

TO
Indicates to whom the privilege is granted.

authorization-name,**...**
Lists one or more authorization IDs. In DB2 UDB for UWO, the authorization ID of the GRANT statement itself cannot be specified.[68]

68. In DB2 UDB for z/OS and OS/390, the CURRENT RULES special register must be set to 'STD' to grant privileges to the authorization ID of the GRANT statement itself.

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, Privileges and Object Ownership" on page 27.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the EXECUTE privilege to others users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the EXECUTE privilege to others unless they have received that authority from some other source.

G In DB2 UDB for UWO, WITH GRANT OPTION is not supported.

Examples

Grant the EXECUTE privilege on PACKAGE PKGA to PUBLIC.

```
GRANT EXECUTE
  ON PACKAGE PKGA
  TO PUBLIC
```

GRANT (Table or View Privileges)

This form of the GRANT statement grants privileges on a table or view.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

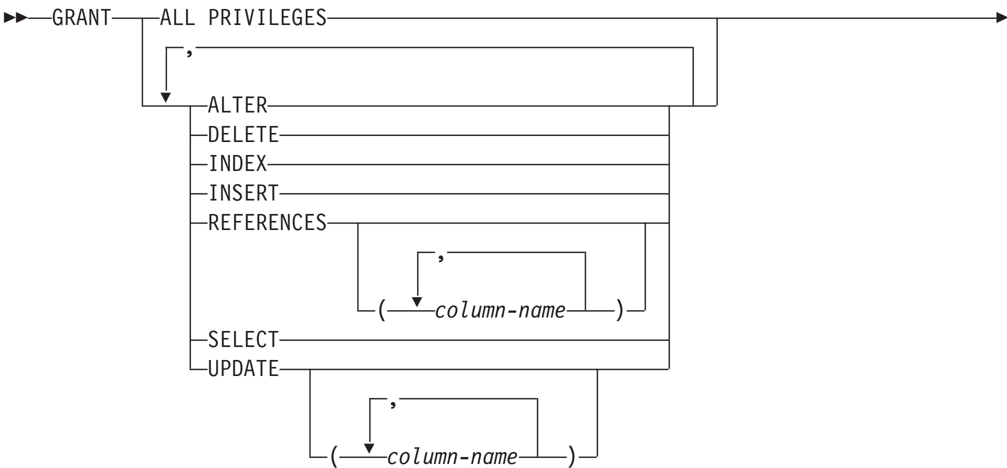
The privileges held by the authorization ID of the statement must include at least one of the following:

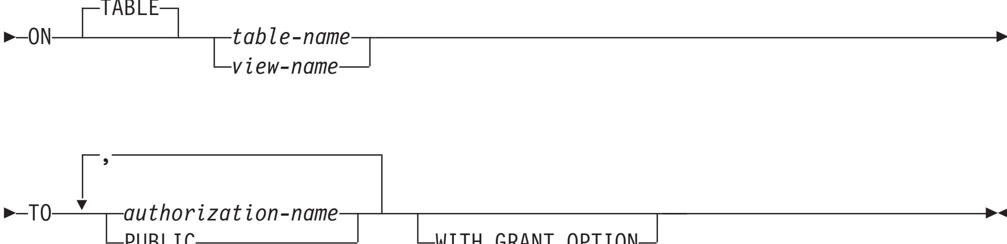
- Ownership of the table or view
- The WITH GRANT OPTION for at least one of the specified privileges. If ALL is specified, the authorization ID must have some grantable privilege on the table or view
- Administrative authority.

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 PRIVILEGES
Grants one or more privileges on the specified table or view. The privileges granted are all those grantable privileges that the authorization ID of the statement has on the identified table or view.

ALTER
Grants the privilege to alter the specified table or create a trigger on the specified table. This privilege cannot be granted on a view.

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 the specified table is a parent. If a list of column names is not specified or if REFERENCES is granted via the specification of ALL PRIVILEGES, the grantee(s) can define referential constraints using all columns of the table as a parent key, even those added later via the ALTER TABLE statement. This privilege cannot be granted on a view.

REFERENCES (*column-name,...***)**
Grants the privilege to add a referential constraint in which the 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 the table identified in the ON clause. This privilege cannot be granted on 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 column names is not specified or if UPDATE is granted via the specification of ALL PRIVILEGES, the grantee(s) can update all updatable columns of the table or view, 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 for the specified table or view 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 the table or view identified in the ON clause. If a view is specified, it must be an updatable view and the specified columns must be updatable columns.

ON *table-name* or *view-name*
Identifies the table or view on which the privileges are granted. The *table-name* or *view-name* must identify a table or view that exists at the current server.

TO
Indicates to whom the privileges are granted.

authorization-name,...
Lists one or more authorization IDs. [69]

PUBLIC
Grants the privilege(s) to a set of users (authorization IDs). For more information, see "Authorization, Privileges and Object Ownership" on page 27.

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the privileges to others users.

If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the privileges to others unless they have received that authority from some other source.

Notes

GRANT rules: The GRANT statement will only grant those privileges that the authorization ID of the statement is allowed to grant.If no privileges were granted, an error is returned.

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
```

69. In DB2 UDB for z/OS and OS/390, the CURRENT RULES special register must be used to grant privileges to the authorization ID of the GRANT statement itself.

INCLUDE

The INCLUDE statement inserts application code, including declarations and statements, into a source program.

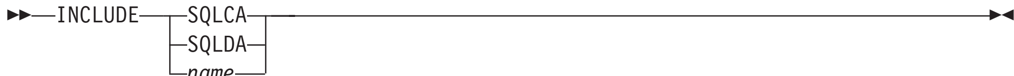
Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in Java or REXX

Authorization

None required.

Syntax



Description

SQLCA
Indicates the description of an SQL communication area (SQLCA) is to be included. INCLUDE SQLCA must not be specified if the program includes a stand-alone SQLSTATE or stand-alone SQLCODE. In COBOL, INCLUDE SQLCA can only be specified within the WORKING-STORAGE SECTION. If INCLUDE SQLCA is not specified in C or COBOL, then the variable SQLSTATE or SQLCODE must appear in the program.

INCLUDE SQLCA must not be specified more than once in the same program. For more information, see “SQL Return Codes” on page 264.

For a description of the SQLCA, see Appendix C, “SQL Communication Area (SQLCA)” on page 525.

SQLDA
Indicates the description of an SQL descriptor area (SQLDA) is to be included. INCLUDE SQLDA can be specified in C and COBOL programs. In COBOL, INCLUDE SQLCA can only be specified within the WORKING-STORAGE SECTION.

For a description of the SQLDA, see Appendix D, “SQL Descriptor Area (SQLDA)” on page 529.

name
Identifies an external file or member containing text that is to be included in the source program being precompiled. In COBOL, INCLUDE *name* must not be specified in other than the DATA DIVISION or PROCEDURE DIVISION.

G G The rules for forming the name and the technique used to map the name to an external file or library member are product-specific.

The included text can contain any statements of the host language and any SQL statements other than INCLUDE statements.

When a program is precompiled, the INCLUDE statement is replaced by source statements.

The INCLUDE statement must be specified at a point in a program where its source statements are allowed.

Examples

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 two 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 fullselect* form is used to insert one or more rows into the table or view using values from the result of the query.

Invocation

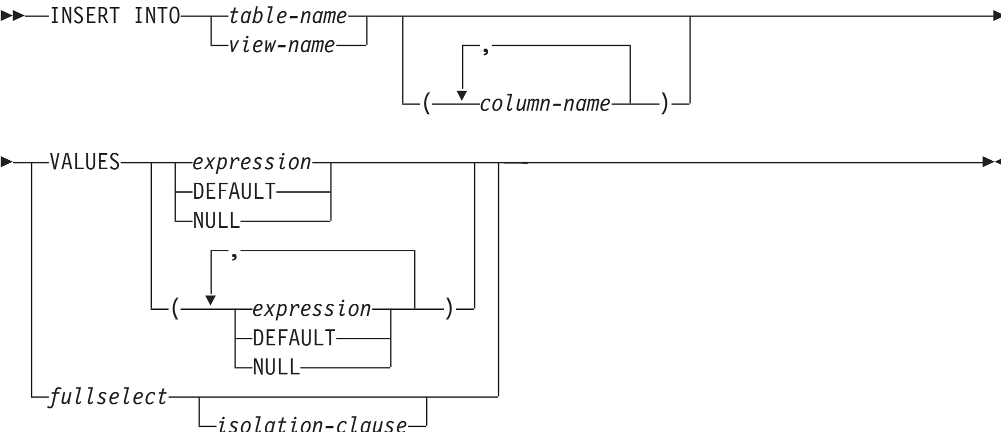
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- The INSERT privilege for the table or view
- Ownership of the table[70]
- Administrative authority.

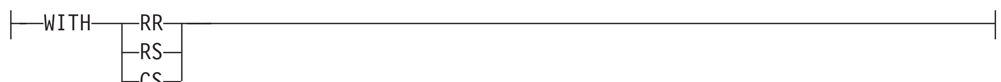
If a *fullselect* is specified, the privileges held by the authorization ID of the statement must also include at least one of the following:
- For every table or view identified in the fullselect:
  – The SELECT privilege on the table or view, or
  – Ownership of the table or view.
- Administrative authority.

Syntax



70. The INSERT privilege on a view is only inherent in administrative authority. Ownership of a view does not necessarily include the INSERT privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

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 376.

(*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 376.

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.

In all the products, SQL statements can be implicitly or explicitly rebound (prepared again). The effect of a rebind on INSERT statements that do not include a column list is as follows:

G G G
- In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, the implicit list of names is reestablished. Therefore, the number of columns into which data is inserted may change.

G G G
- In DB2 UDB for iSeries, the list of names is not reestablished. Therefore, the number of columns into which data is inserted by the statement does not change.

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 variable that is declared in the program in accordance with the rules for declaring host structures and variables. In the operational form of the statement, a reference to a 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 85 and "Host Structures" on page 89.

The number of values in the VALUES clause must equal the number of names in the implicit or explicit column list. The first value is inserted into the first column in the list, then the second value into the second column, and so on.

expression
Any expression of the type described in "Expressions" on page 96, that does not include a column name. If *expression* is a *host-variable*, the host variable can identify a structure.

DEFAULT
The default value assigned to the column. DEFAULT can be specified only

for columns that have a default value. For information on default values of data types, see the description of the DEFAULT clause for CREATE TABLE in “CREATE TABLE” on page 353.

NULL
Specifies the null value as the value of the column. Specify NULL only for nullable columns.

fullselect
Specifies a set of new rows in the form of the result table of a fullselect. If the result table is empty, SQLSTATE is set to ‘02000’.

For an explanation of fullselect, see “fullselect” on page 248.

When the base object of the INSERT and the base object of the fullselect or any subquery of the fullselect, are the same table, the fullselect is completely evaluated before any rows are inserted.

The number of columns in the result table must equal the number of names in the column list. The value of the first column of the result is inserted in the first column in the list, then the second value in the second column, and so on.

G In DB2 UDB for z/OS and OS/390, if the object table is self-referencing, the
G fullselect must not return more than one row.

isolation-clause
Specifies the isolation level used by the statement.

WITH

Introduces the isolation level, which may be one of:
- RR Repeatable read
- RS Read stability
- CS Cursor stability

If *isolation-clause* is not specified, the default isolation is used. See “Isolation Level” on page 13 for a description of how the default is determined.

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 the insert is into a view, the default value is inserted into any column of the base table that is not included in the view. Hence, all columns of the base table that are not in the view must have a default value.

Assignment: Insert values are assigned to columns in accordance with the storage assignment rules described in “Assignments and Comparisons” on page 58.

Validity: Inserts must obey the following rules. If they do not, or if any other errors occur during the execution of the INSERT statement, no rows are inserted.

- *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 inserted into 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 insert, this would occur after all rows were inserted.

- *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 inserted into the table (SQLSTATE 23513).

  All check constraints are effectively validated at the end of the statement. In the case of a multiple-row insert, this would occur after all rows were inserted.
- *Views and the CHECK OPTION clause:* If a view is identified, the inserted rows must conform to any applicable CHECK OPTION clause (SQLSTATE 44000). For more information, see "CREATE VIEW" on page 376.

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 return error conditions based on the insert values.

Referential Integrity: Each nonnull insert value of a foreign key must be equal to some value of the parent key of the parent table in the relationship (SQLSTATE 23503).

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.

Notes

Insert operation errors: If an insert value violates any constraints, or if any other error occurs during the execution of the INSERT statement, changes from this statement and any triggered SQL statements are rolled back.

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: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful INSERT statement. Until these 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 isolation level UR through a read-only cursor, SELECT INTO statement, or subquery.

The locks can prevent other application processes from performing operations on the table.

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) };
```

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:

The SELECT privilege for the table

Ownership of 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 it must not identify a catalog table.

IN SHARE MODE
: Prevents concurrent application processes from executing any but read-only operations on the table.

IN EXCLUSIVE MODE
: Prevents concurrent application processes from executing any operations on the table. This may or may not apply to concurrent application processes running at isolation level UR. The rule is product-specific.

G

Notes

Locks Obtained: Locking is used to prevent concurrent operations. A lock is not necessarily acquired during the execution of the LOCK TABLE statement if a suitable lock already exists. The lock that prevents concurrent operations is held until the end of the unit of work.

Examples

Request an exclusive lock on the DEPARTMENT table.

```
LOCK TABLE DEPARTMENT IN EXCLUSIVE MODE
```

OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

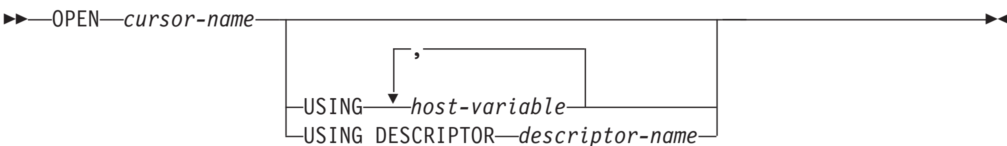
Invocation

This statement can be embedded only 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 381 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 "Notes" on page 382. 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 was not successfully prepared, or is not a *select-statement*, an error is returned.

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 either be derived during the execution of the OPEN statement (in which case a temporary table is created for 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 state of the cursor is effectively 'after the last row.' An empty table does not cause an SQLSTATE warning of '02000' when the OPEN statement is executed. A subsequent fetch for the cursor may return the SQLSTATE warning of '02000'.

USING
Introduces the values that are substituted for the parameter markers (question marks) of a prepared statement. For an explanation of parameter markers, see "PREPARE" on page 433. If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, USING must be used. If the prepared statement does not include parameter markers, USING is ignored.

host-variable,...
Identifies 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 resulting 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:

- SQLN to indicate the number of SQLVAR entries 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 entries to indicate the attributes of the variables.

Note that the rules for REXX are different. For more information see Appendix K, “Coding SQL Statements in REXX Applications” on page 643.

The SQLDA must have enough storage to contain all occurrences of SQLVAR entries. If an SQLVAR entry includes a LOB or distinct type based on a LOB, there must be additional SQLVAR entries for each parameter. For more information on the SQLDA, which includes a description of the SQLVAR and an explanation on how to determine the number of SQLVAR entries, see Appendix D, “SQL Descriptor Area (SQLDA)” on page 529.

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.

Notes

Closed state of cursors: Cursors are in an open state after a successful OPEN statement. The state of the cursor becomes closed in many ways:

- All cursors in a program are in a closed state when the program is first called.
- Cursors declared not using the WITH HOLD option are in a closed state when a unit of work is committed.
- Cursors declared in a procedure not using the WITH RETURN clause may be closed when the procedure returns. For more information, see “DECLARE CURSOR” on page 381.
- Cursors are in a closed state when a unit of work is rolled back.

A cursor can also be in the closed state because:

- A CLOSE statement was executed.
- An error was detected that made the position of the cursor unpredictable.
- 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, the database manager 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.
- 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 CUR is positioned on a row of its result table defined as SELECT * FROM T, and a row is inserted into T, the effect of that insert on the result table is not predictable because its rows are not ordered. A subsequent FETCH CUR 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 the value of 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 36 on page 435.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P using storage assignment rules as described in “Assignments and Comparisons” on page 58. Thus:

- V must be compatible with the target.
- If V is a string, its length (including trailing blanks) must not be greater than the length attribute of the target.
- If V is a number, the whole part of the number must not be truncated.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.
- If the target cannot contain nulls, V must not be null.

When the SELECT statement of the cursor is evaluated, 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 CUR 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 CUR before the first row to be fetched.

```
EXEC SQL  DECLARE CUR CURSOR FOR
            SELECT DEPTNO, DEPTNAME, MGRNO FROM DEPARTMENT
              WHERE ADMRDEPT = 'A00'  END-EXEC.

EXEC SQL  OPEN CUR  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 long hv_int;
   char hv_vchar64[65];
   char stmt1_str[200];
EXEC SQL  END DECLARE SECTION;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;

EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING :hv_int, :hv_vchar64;
```

Example 3: Code an OPEN statement as in example 2, but in this case the number and data types of the parameter markers in the WHERE clause are not known.

```
EXEC SQL  BEGIN DECLARE SECTION;
   char stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str INTO :sqlda;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

/* Set up the SQLDA */

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 Chapter 4, "Queries" on page 233 for the authorization rules that apply when a SELECT statement is prepared. The authorization ID is the runtime authorization ID.

Syntax

Description

statement-name
Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

INTO
If INTO is used, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by *descriptor-name*. Thus, the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO :SQLDA FROM :V1;
```

is logically 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 529. 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 Appendix K, "Coding SQL Statements in REXX Applications" on page 643.):

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 Occurrences of SQLVAR Entries are Needed" on page 531.

See "DESCRIBE" on page 391 for an explanation of the information that is placed in the SQLDA.

FROM
Introduces the statement string. The statement string is the value of the specified *host-variable*.

host-variable
Identifies the *host-variable* that contains the statement string. The *host-variable* must identify a host variable that is described in the application program in accordance with the rules for declaring character string variables. The host variable must not have a CLOB data type, and an indicator variable must not be specified. In COBOL, the host variable must be a varying-length string variable. In C, the host variable must not be a NUL-terminated string.

The statement string must be one of the following SQL statements:

ALTER	DROP	REVOKE
COMMENT	GRANT	ROLLBACK
COMMIT	INSERT	select-statement
CREATE	LOCK TABLE	SET PATH
DELETE	RENAME	UPDATE

The statement string must not:
- Begin with EXEC SQL.
- End with END-EXEC or a semicolon.
- Include references to host variables.
- Include comments.

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 429 and "EXECUTE" on page 401.

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 invocation of a CAST specification is a "promise" that the data type of the parameter at run time will be of the data type specified or some data type that is assignable 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 data

type that can be converted to VARCHAR(12). For more information, refer to "Assignments and Comparisons" on page 58.

Untyped parameter marker
A parameter marker that is specified without its target data type. It has the form of a single question mark. The data type of an untyped parameter marker is provided by context. For example, the untyped parameter marker in the predicate of the above update statement is the same as the data type of the EMPNO column.

Typed parameter markers can be used in dynamic SQL statements wherever a host variable is supported and the data type is based on the promise made in the CAST function.

Untyped parameters markers can be used in dynamic SQL statements only in selected locations where host variables are supported. These locations and the resulting data type are found in Table 36. The locations are grouped in this table into expressions, predicates and functions to assist in determining applicability of an untyped parameter marker.

Table 36. 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	The data type of the other operand of the subquery. [71]
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. [71]
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.)	Error
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 side of a SET clause of an UPDATE statement.	The data type of the column. If the column is defined as a distinct type, then it is the source data type of the distinct type. [71]
The expression following the CASE keyword in a simple CASE expression	Error

Table 36. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
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 68 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 68 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 distinct type, then it is the source data type of the distinct type.[71]
As a value on the right side of a SET special register statement	The data type of the special register.
As a value in the INTO clause of the VALUES INTO statement	The data type of the associated expression. [71]
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. [71]
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 68 on all operands that are other than untyped parameter markers.
All operands of an IN predicate, for example, ? IN (?,?,?)	Error
The 1st operand of an IN predicate where the right side is a subselect, for example, ? IN (subselect).	Data type of the selected column
The 1st operand of an IN predicate where the right 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 68 on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers.

Table 36. Untyped Parameter Marker Usage (continued)

Untyped Parameter Marker Location	Data Type
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 68 on all operands of the IN predicate (operands to the left and right of the IN predicate) that are other than untyped parameter markers. G In DB2 UDB for z/OS and OS/390, the operand to the left of the IN predicate are not included.
All three operands of the LIKE predicate.	Error
The match expression of the LIKE predicate.	Error
G The pattern expression of the LIKE predicate.	Either VARCHAR(n) or VARGRAPHIC(n) or BLOB(n) depending on the data type of the match expression, where n is product-specific. For information on using fixed-length host variables for the value of the pattern see "LIKE Predicate Notes" on page 122.
The escape expression of the LIKE predicate.	Either VARCHAR(n) or VARGRAPHIC(1) or BLOB(1) depending on the data type of the match expression, where n is 1 or 2 depending on the default CCSID.
Operand of the NULL predicate.	Error
Functions	
All operands of COALESCE, NULLIF, or VALUE.	Error
Any operand of COALESCE, NULLIF, or VALUE where at least one operand is other than an untyped parameter marker.	Result of applying the "Rules for Result Data Types" on page 68 on all operands that are other than untyped parameter markers.
G The second operand of POSSTR.	Either VARCHAR(n) or VARGRAPHIC(n) or BLOB(n) depending on the data type of the other operand, where n is product-specific.
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 an error is returned.

G A product-specific option may be used to 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:

71. If the data type is DATE, TIME, or TIMESTAMP, then CHAR(n), where n is product-specific.

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:[72]

- A CONNECT (Type 1) statement is executed.
- A prepared statement is associated with a release-pending connection and a successful commit occurs.

G In DB2 UDB for z/OS and OS/390, all prepared statements are destroyed when
G the unit of work ends except:

- G if the SELECT statement whose cursor is declared with the option WITH HOLD G persists over the execution of a commit operation if the cursor is open when the G commit operation is executed
- G if SELECT, INSERT, UPDATE, and DELETE statements that are bound with G KEEPDYNAMIC(YES).

Scope of a Statement: The scope of *statement-name* is the source program in which it is defined. A prepared statement can only be referenced 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.

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.

Examples

Example 1: Prepare and execute a statement other than a 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.

72. Prepared statements may be cached and not actually destroyed. However, a cached statement can only be used if the same statement is prepared again.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER  END-EXEC.

EXEC SQL  EXECUTE STMT_NAME  END-EXEC.
```

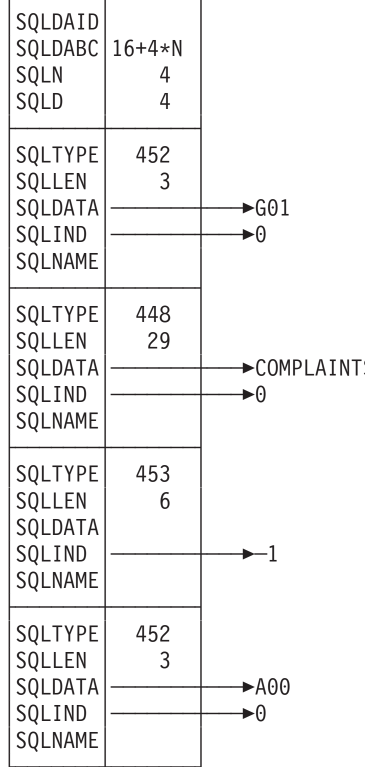
Example 2: Prepare and execute a non-select-statement as in example 1, except assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER  END-EXEC.

/* Set up the SQLDA */

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 issuing the EXECUTE statement.



RELEASE (Connection)

The RELEASE statement places one or more connections in the release-pending state.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

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 with a length attribute that is not greater than 18. In DB2 UDB for z/OS and OS/390, the maximum length of the value is 16. In DB2 UDB for UWO, the maximum length of the value is 8.
- It must not be followed by an indicator variable
- If a reserved word is used as an identifier in SQL, it must be specified in uppercase and either as a delimited identifier or in a host variable.
- 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 application process.

CURRENT
: Identifies the current connection of the application process. The application process must be in the connected state.

An application server named CURRENT can only be identified by a host variable or a delimited identifier.

ALL or ALL SQL
: Identifies all existing connections of the application process (local as well as remote connections).

An error or warning does not occur if no connections exist when the statement is executed.

An application server named ALL can only be identified by a host variable or a delimited identifier.

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 operation. If the RELEASE statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.

Notes

RELEASE and CONNECT (Type 1): Using CONNECT (Type 1) semantics does not prevent using RELEASE.

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 application process 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 default application 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 WITH HOLD.

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 TOROLAB;
```

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 them to be ended during the next commit operation

```
EXEC SQL RELEASE ALL;
```

RENAME

The RENAME statement renames an existing table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table
- Adminstrative authority.

Syntax

Description

table-name
: Identifies the existing table that is to be renamed. The name, including the implicit or explicit qualifier, must identify a table that exists at the current server.

  The table must not be:

  - referenced in any existing view definitions
  - referenced in triggered statements, and cannot have a trigger defined on it
  - a catalog table
  - a parent or dependent table in any referential integrity constraints
  - defined with any check constraints.

new-table-identifier
: Specifies the new name for the table without a qualifier. The qualifier of the *table-name* is used to qualify the new identifier for the table. The new qualified name must not identify a table, view, alias, or index that already exists at the current server.

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 9.

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.

Examples

Change the name of the EMPLOYEE table to CURRENT_EMPLOYEES:

```
RENAME TABLE  EMPLOYEE
  TO CURRENT_EMPLOYEES
```

REVOKE (Distinct Type Privileges)

This form of the REVOKE statement revokes the privilege 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.

G In DB2 UDB for UWO, this statement is not supported. Instead, PUBLIC implicitly
G has the USAGE privilege on all distinct types which cannot be revoked.

Authorization

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

```
>>--REVOKE--USAGE--ON--DISTINCT TYPE--distinct-type-name---------------------->

            .-,------------------.
            V                    |
>--FROM---+-authorization-name-+-+-----------------------------------------><
          '-PUBLIC-------------'
```

Description

USAGE
Revokes the privilege to use a distinct type. The privilege may not be revoked if the authorization ID of the statement did not grant the USAGE privilege on the distinct type. For more information see, "Authorization, Privileges and Object Ownership" on page 27.

A user with administrative authority may revoke the USAGE privilege granted
G by others. The technique is product-specific.

ON DISTINCT TYPE *distinct-type-name*
Identifies the distinct type from which the privilege is revoked. The *distinct-type-name* must identify a distinct type that exists at the current server.

FROM
Identifies from whom the privilege is revoked.

authorization-name,**...**
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, Privileges and Object Ownership" on page 27.

Notes

REVOKE Restrictions: The USAGE privilege must not be revoked on a distinct type if:

- The *authorization-name* owns a function or procedure that uses the distinct type, or
- The *authorization-name* owns a table that has a column that uses the distinct type.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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.

Examples

Revoke the USAGE privilege on distinct type SHOESIZE from user JONES.

```
REVOKE USAGE
  ON DISTINCT TYPE SHOESIZE
  FROM JONES
```

REVOKE (Function or Procedure Privileges)

This form of the REVOKE statement revokes the privilege 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.

G
G
G

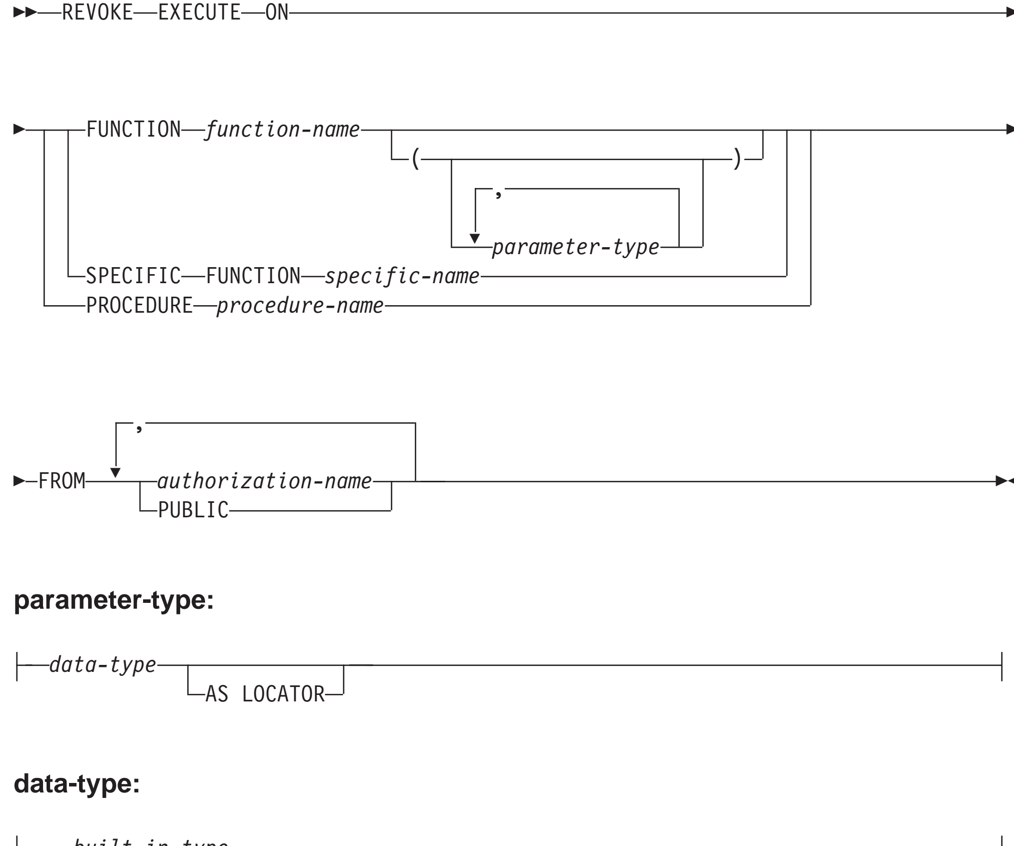
In DB2 UDB for UWO, this statement is not supported. Instead, PUBLIC implicitly has the EXECUTE privilege on all functions and procedures which cannot be revoked.

Authorization

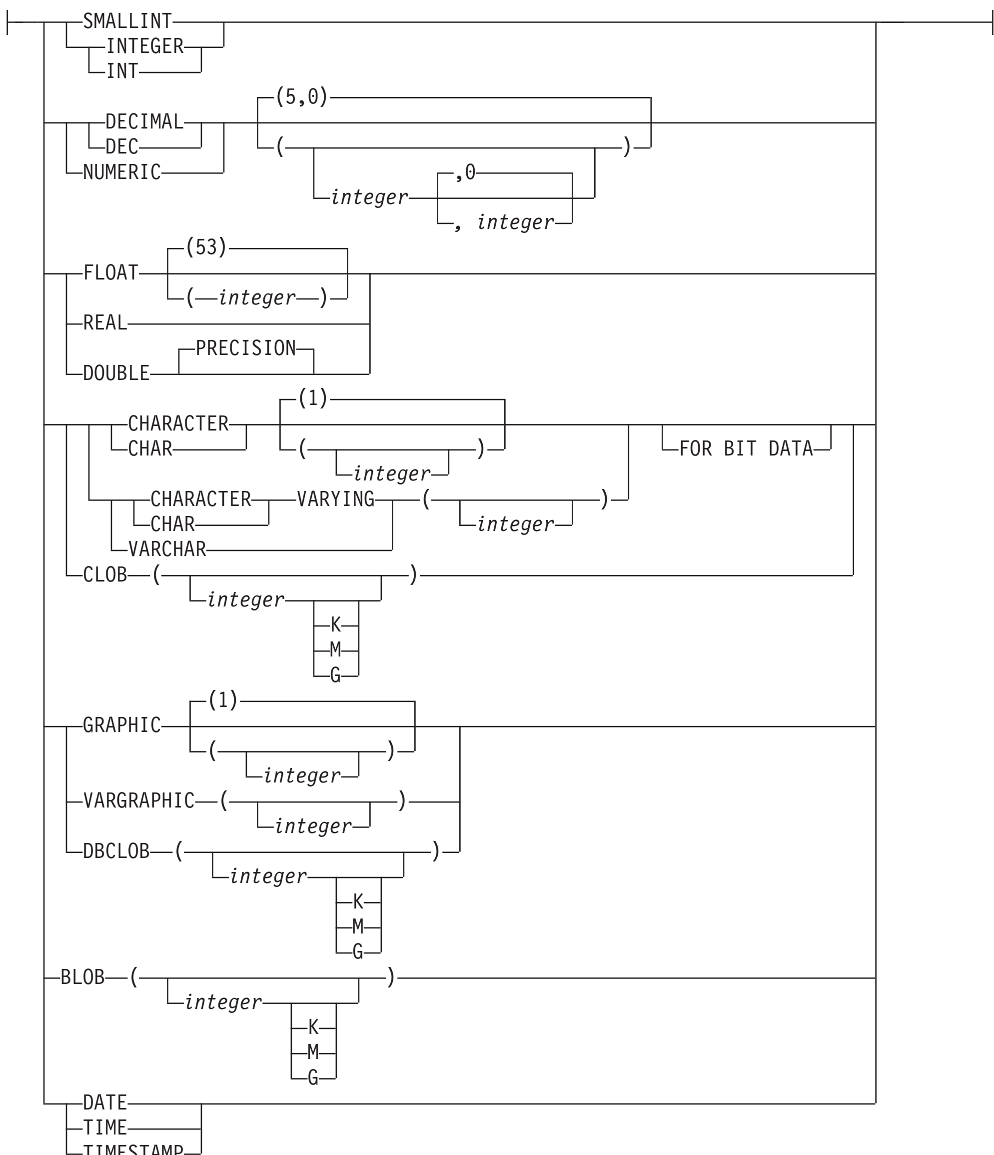
The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the function or procedure
- Administrative authority.

Syntax



built-in-type:



Description

EXECUTE

Revokes the privilege to execute a function or procedure. The privilege may not be revoked if the authorization ID of the statement did not grant the EXECUTE privilege on the function or procedure. For more information see, “Authorization, Privileges and Object Ownership” on page 27.

G A user with administrative authority may revoke the EXECUTE privilege granted by others. The technique is product-specific.

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 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 from 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 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.

AS LOCATOR
Specifies that the function is defined to receive a locator for this parameter. If AS LOCATOR is specified, the data type must be a LOB or a distinct type based on a LOB.

SPECIFIC FUNCTION *specific-name*
Identifies the function by its specific name. The *specific-name* must identify a specific function that exists at the current server.

PROCEDURE *procedure-name*
Identifies the procedure from which the privilege is revoked. The *procedure-name* must identify a procedure that exists at the current server.

FROM
Identifies from whom the privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, Privileges and Object Ownership" on page 27.

Notes

REVOKE Restrictions: The EXECUTE privilege must not be revoked on a function or procedure if the *authorization-name* owns any of the following objects:

- A function that is sourced on the function
- A view that uses the function
- A trigger that uses the function or procedure
- A table that uses the function in a DEFAULT clause

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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.

Examples

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 revokes the privilege to execute statements in a package.

Invocation

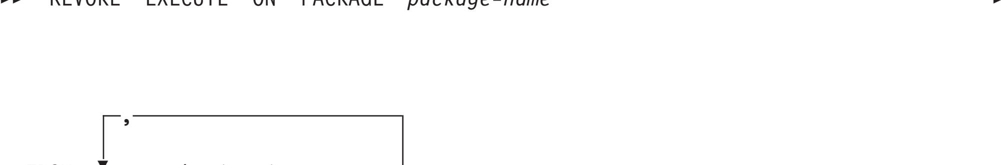
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the package
- Administrative authority.

Syntax



Description

EXECUTE
Revokes the privilege to execute SQL statements in a package. The privilege may not be revoked if the authorization ID of the statement did not grant the EXECUTE privilege on the package. For more information see, "Authorization, Privileges and Object Ownership" on page 27.

G A user with administrative authority may revoke the EXECUTE privilege granted by others. The technique is product-specific.

ON PACKAGE *package-name*
Identifies the package from which the EXECUTE privilege is revoked. The *package-name* must identify a package that exists at the current server.

FROM
Identifies from whom the privilege is revoked.

authorization-name,...
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of the privilege to PUBLIC. For more information, see "Authorization, Privileges and Object Ownership" on page 27.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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.

Examples

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 and View Privileges)

This form of the REVOKE statement revokes privileges on a table or view.

Invocation

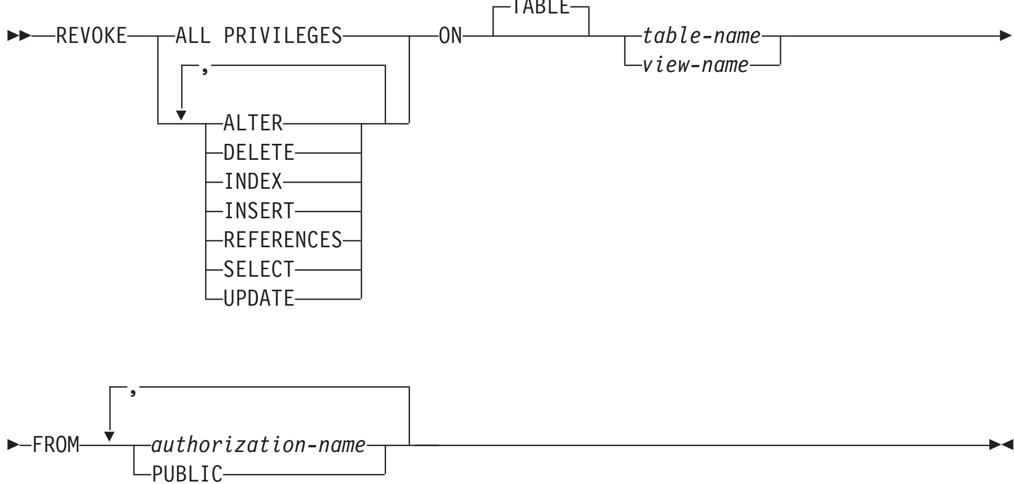
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table or view
- Administrative authority.

Syntax



Description

Each keyword revokes the privilege described, but only as it applies to the table or view named in the ON clause. The same keyword must not be specified more than once.

A privilege may not be revoked if the authorization ID of the statement did not grant the specified privilege on the table or view. For more information see, "Authorization, Privileges and Object Ownership" on page 27.

G A user with administrative authority may revoke privileges granted by others. The
G technique is product-specific.

ALL PRIVILEGES
Revokes one or more privileges from each *authorization-name*. The privileges revoked are all those privileges on the identified table or view which were granted to the *authorization-name*.

ALTER
Revokes the privilege to alter the specified table or create a trigger on the specified table.

DELETE
Revokes the privilege to delete rows from the specified table or view.

INDEX
Revokes the privilege to create an index on the specified table.

INSERT
Revokes the privilege to insert rows in the specified table or view.

REFERENCES
Revokes the privilege to add a referential constraint in which the specified table is the parent.

SELECT
Revokes the privilege to create a view or read data from the specified table or view.

UPDATE
Revokes the privilege to update rows in the specified table or view.

ON *table-name* or *view-name*
Identifies the table or view from which the privileges are revoked. The *table-name* or *view-name* must identify a table or view that exists at the current server.

FROM
Identifies from whom the privileges are revoked.

authorization-name,**...**
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once.

PUBLIC
Revokes a grant of privileges to PUBLIC. For more information, see "Authorization, Privileges and Object Ownership" on page 27.

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 the privilege itself and then grant it again without specifying WITH GRANT OPTION.

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.

Effect on views and access plans: Revoking a privilege that was used to create a view might cause the view to be dropped. Revoking a privilege that was used to
G create an access plan may invalidate the access plan. In both cases, the rules are
G product-specific.

G **Dependent privileges:** In DB2 UDB for z/OS and OS/390, when a privilege is
G revoked from a user, every privilege dependent on that privilege is also revoked.
G For more information see the DB2 UDB for z/OS and OS/390 product books. In
G DB2 UDB for iSeries and DB2 UDB for UWO, privileges are not dependent upon
G other privileges.

Examples

Example 1: Revoke SELECT privileges on table EMPLOYEE from user ENGLES.

```
REVOKE SELECT
  ON  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  EMPLOYEE
  FROM PUBLIC
```

Example 3: Revoke all privileges on table EMPLOYEE from users PELLOW and ANDERSON.

```
REVOKE ALL
  ON  EMPLOYEE
  FROM PELLOW, ANDERSON
```

ROLLBACK

The ROLLBACK statement ends a unit of work and backs out the database changes that were made in 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.

Authorization

None required.

Syntax

Description

The ROLLBACK statement ends the unit of work in which it is executed. It backs out all changes made by SQL schema statements and SQL data change statements during the unit of work. For more information see Chapter 5, “Statements” on page 259.

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.

Effect of rollback: Rollback causes the following to occur:

- All cursors that were opened during the unit of work are closed.
- All LOB locators are freed.
- All locks acquired by the unit of work are released.
- For DB2 UDB for z/OS and OS/390, all statements that were prepared during the unit of work are destroyed. 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.

Other transaction environments: SQL ROLLBACK may not be available in other transaction environments, such as IMS and CICS. To do a rollback operation in these environments, SQL programs must use the call prescribed by their transaction manager.

Examples

See “Examples” on page 295 under COMMIT which shows the use of the ROLLBACK statement.

SELECT

The SELECT statement is a form of query. It can be embedded in an SQLJ application program or issued interactively. For detailed information, see “select-statement” on page 250 and Chapter 4, “Queries” on page 233.

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.

Invocation

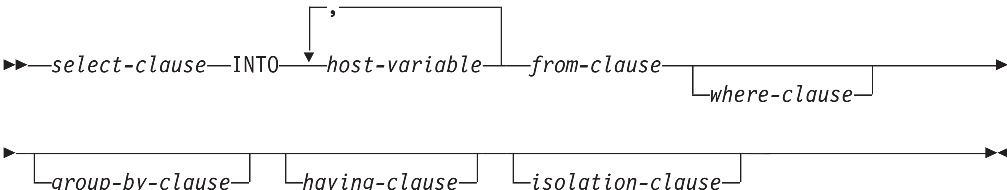
This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in REXX.

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, one of the following:
  - The SELECT privilege on the table or view
  - Ownership of the table or view
- Administrative authority.

Syntax



Description

The result table is derived by evaluating the *isolation-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, and *select-clause*, in this order.

See Chapter 4, "Queries" on page 233 for a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-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 INTO, 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.

Notes

Host Variable Assignment: Each assignment to a host variable is performed according to the retrieval assignment rules described in "Assignments and Comparisons" on page 58. 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 85.

If an assignment error occurs, the values in the host variables are unpredictable.

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: 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.[73]

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 an error is returned. See “Datetime Assignments” on page 62 for details.

Examples

Example 1: Using a COBOL program statement, put the maximum salary (SALARY) from the EMPLOYEE table into the host variable MAX-SALARY.

```
EXEC SQL  SELECT MAX(SALARY)
            INTO :MAX-SALARY
            FROM EMPLOYEE
END-EXEC.
```

Example 2: Using a Java program statement, select the row from the EMPLOYEE table on the connection context ’ctx’ with a 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 (java.lang.String) and HOST_EDUCATE (java.lang.Integer).

```
#sql [ctx] { SELECT LASTNAME, EDLEVEL
            INTO :HOST_NAME, :HOST_EDUCATE
            FROM EMPLOYEE
            WHERE EMPNO = :HOST_EMP };
```

73. In DB2 UDB for UWO, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

SET CONNECTION

The SET CONNECTION statement establishes the current server of the process by identifying one of its existing connections.

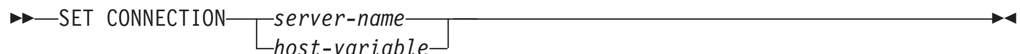
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It must not be specified in Java or REXX.

Authorization

None required.

Syntax



Description

server-name or *host-variable*
: Identifies the connection by the specified server name or the server name contained in the host variable. If *host-variable* is specified:

  - It must be a character-string variable with a length attribute that is not greater than 18. In DB2 UDB for z/OS and OS/390, the maximum length of the value is 16. In DB2 UDB for UWO, the maximum length of the value is 8.
  - 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.

G
G
G

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 application server S is placed in the SQLERRP field of the SQLCA. The format below applies if the application server is a DB2 Universal Database product. The information has the form *pppvvrrm*, where:
  - *ppp* is:

    DSN for DB2 UDB for z/OS and OS/390

    QSQ for DB2 UDB for iSeries

    SQL for DB2 UDB for UWO

  - *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 z/OS and OS/390, the value would be 'DSN07010'.
- Additional information about the connection is placed in the SQLERRMC field of the SQLCA. The contents are product-specific.
- Any previously current connection is placed in the dormant state.

G

If the SET CONNECTION statement is unsuccessful, the connection state of the application process 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 application process.

Examples

Execute SQL statements at TOROLAB, execute SQL statements at STLLAB, and then execute more SQL statements at TOROLAB.

```
EXEC SQL CONNECT TO TOROLAB;

  (execute statements referencing objects at TOROLAB)

EXEC SQL CONNECT TO STLLAB;

  (execute statements referencing objects at STLLAB)

EXEC SQL SET CONNECTION TOROLAB;

  (execute statements referencing objects at TOROLAB)
```

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 PATH

The SET PATH statement changes the value of the CURRENT PATH special register.

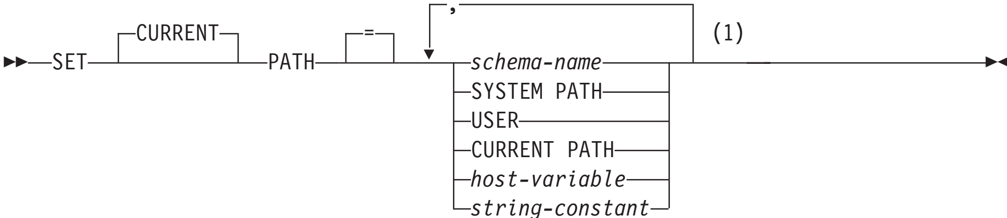
Invocation

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



Notes:

1 SYSTEM PATH, USER, and CURRENT PATH may each be specified at most once on the right side of the statement.

Description

schema-name
: Identifies a schema. No validation that the schema exists is made at the time the SQL 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
: Specifies the schema names for the system path for the platform.

USER
: Specifies the value of 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:

- Must be a CHAR or VARCHAR variable. The actual length of the contents of the *host-variable* must not exceed the length of a schema name. See Appendix A, "SQL Limits" on page 509.
- Must not be followed by an indicator variable.
- Must not be the null value.

- Must include a schema name that is left justified and conforms to the rules for forming an ordinary identifier. The schema name must not be specified as a 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
Specifies a string contant that contains a schema name. The string constant:

- Must have a length that does not exceed the maximum length of a schema name. See Appendix A, “SQL Limits” on page 509.
- Must include a schema name that is left justified and conforms to the rules for forming an ordinary identifier. The schema name must not be specified as a delimited identifier.
- Must not contain lowercase letters or characters that cannot be specified in an ordinary identifier.
- Must not contain SYSTEM PATH, USER, or CURRENT PATH.

Notes

Transaction Considerations: The SET PATH statement is not a commitable operation. ROLLBACK has no effect on the SQL path.

Rules for the Content of the SQL path:

- A schema name cannot appear more than once in the SQL 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, changing each double quote character to two double quote characters within the schema name as necessary, and then separating each schema name by a comma. The length of the resulting string cannot exceed 254. See Appendix A, “SQL Limits” on page 509 for more information.
- A schema name that does not conform to the rules for an ordinary identifier (for example: a schema name that contains lowercase characters or characters that cannot be specified in an ordinary identifier), must be specified as a *delimited schema name* and must not be specified within a host variable or string constant. If the schema name changes dynamically in the application then the SET PATH statement must be dynamically prepared using the PREPARE and EXECUTE statements rather than changing the content of a host variable.
- There is a difference between specifying a single keyword (such as USER, or 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 on DB2 UDB for z/OS and OS/390, assuming that the current value of the USER special register is SMITH, then SET PATH = SYSIBM, SYSPROC, USER, ″USER″ results in a CURRENT PATH value of ″SYSIBM″,″SYSPROC″,″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.

G G **The System Path:** SYSTEM PATH refers to the system path for a platform. The schemas in the *system path* are platform-specific.

G G G
- For DB2 UDB for z/OS and OS/390, the schemas of the system path are ″SYSIBM″, ″SYSFUN″, and ″SYSPROC″, and SYSTEM PATH is the same as specifying ″SYSIBM″, ″SYSFUN″, ″SYSPROC″.

G G
- For DB2 UDB for UWO, the schemas of the system path are ″SYSIBM″ and ″SYSFUN″, and SYSTEM PATH is the same as specifying ″SYSIBM″, ″SYSFUN″.

G G
- For DB2 UDB for iSeries, the schemas of the system path are ″QSYS″ and ″QSYS2″, and SYSTEM PATH is the same as specifying ″QSYS″, ″QSYS2″.

When using the SET PATH statement, the system path must be specified explicity using SYSTEM PATH or implicitly by using CURRENT PATH which already includes the system path.

Using the SQL path: The CURRENT PATH special register specifies the SQL path used to resolve user-defined distinct types, functions and procedures in dynamic SQL statements. See “SQL Path” on page 38.

Examples

Example 1: 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 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.

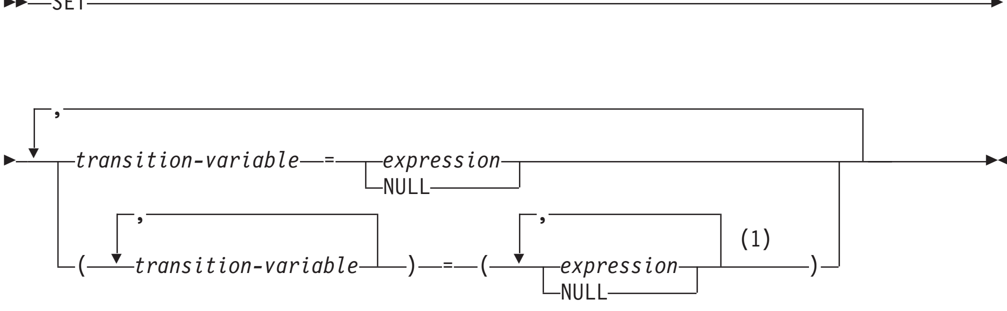
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- The UPDATE privilege on any column associated with a *transition-variable* that occurs on the left side of the SET transition-variable statement in a BEFORE trigger
- Administrative authority.

The privileges required to set a *transition-variable* is checked at the time the trigger is created. For more information, see “CREATE TRIGGER” on page 368.

Syntax



Notes:

1 The number of *expressions* and NULLs 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 storage assignment rules. For more information see “Assignments and Comparisons” on page 58.

expression
Indicates the new value of the *transition-variable*. The expression is any expression of the type described in "Expressions" on page 96 that does not include a column function.

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.

Notes

Multiple Assignments: If more than one assignment is included in the same SET clause, 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
```

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 can 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:

- The UPDATE privilege for the table or view
- The UPDATE privilege on each column to be updated
- Ownership of the table [74]
- Administrative authority.

If the right side of *assignment-clause* contains a reference to a column of the table or view, or if *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:

- The SELECT privilege for the table or view [75]
- Ownership of the table or view
- Administrative authority.

If the statement includes a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:

- For every table or view identified in the subquery:
  - The SELECT privilege on the table or view, or
  - Ownership of the table or view
- Administrative authority.

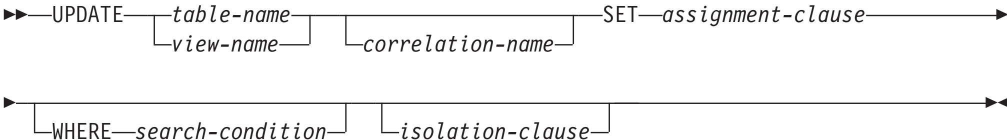
For more information about the subquery authorization rules, see Chapter 4, "Queries" on page 233.

74. The UPDATE privilege on a view is only inherent in administrative authority. Ownership of a view does not necessarily include the UPDATE privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

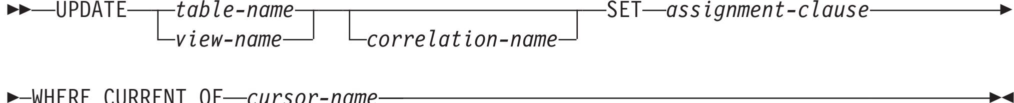
75. In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, the authorization ID of the statement only requires the UPDATE privilege for the table or view. To require the SELECT privilege, a standards option must be in effect. For DB2 UDB for z/OS and OS/390 use the program preparation option SQLRULES(STD) or set the CURRENT RULES special register to 'STD'. For DB2 UDB for UWO, use the program preparation option LANGLEVEL SQL92E.

Syntax

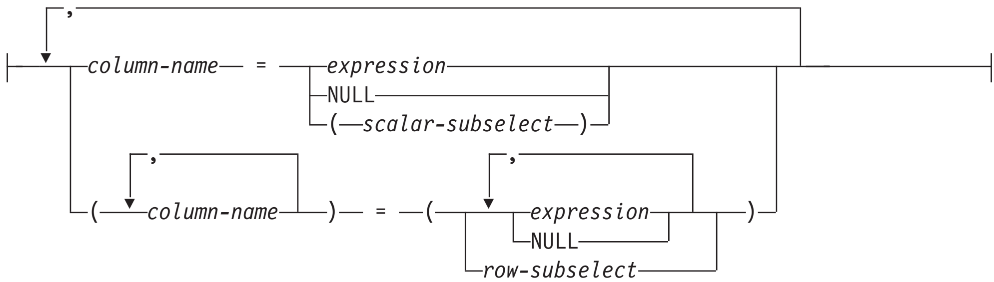
Searched UPDATE:



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 view that is not updatable. For an explanation of updatable views, see "CREATE VIEW" on page 376.

correlation-name
Can be used within *search-condition* or *assignment-clause* to designate the table or view. For an explanation of *correlation-name*, see "Correlation Names" on page 79.

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

column of the specified table or view, and that column must be an updatable column. The column names must not be qualified, and a column name must not be specified more than once.

For a Positioned UPDATE:

- If the FOR UPDATE clause was specified in the *select-statement* of the cursor, each *column-name* must also appear in the FOR UPDATE clause.
- If the FOR UPDATE clause was not specified in the *select-statement* of the cursor, the name of any updatable column may be specified.[76]

For more information, see "update-clause" on page 254.

A view column derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same UPDATE statement.

expression
Indicates the new value of the column. The *expression* is any expression of the type described in "Expressions" on page 96, that does not include a column function.

A *column-name* in an expression must name a column of the named table or view. For each updated row, the value of the column in the expression is the value of the column in the row before the row is updated.

NULL
Specifies the null value as the new value of the column. Specify NULL only for nullable columns.

scalar-subselect
Specifies a subselect that returns a single result row with a single column. The column value is assigned to the corresponding *column-name*. If the subselect returns no rows, the null value is assigned.

For a positioned update, if the table or view that is the object of the UPDATE statement is used in the subselect, the column from the instance of the table or view in the subselect cannot be the same as *column-name*, the column being updated.

The subselect must not contain a GROUP BY or HAVING clause. If the subselect refers to columns to be updated, the value of such a column in the subselect is the value of the column in the row before the row is updated.

row-subselect
Specifies a subselect that returns a single result row. The number of columns in the row must match the number of *column-names* that are specified. The column values are assigned to each corresponding *column-name*. If *row-subselect* returns no rows, null values are assigned.

The subselect must not contain a GROUP BY or HAVING clause. If the subselect refers to columns to be updated, the value of such a column in the subselect is the value of the column in the row before the row is updated.

76. In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, a program preparation option must be used if the UPDATE clause is not specified and the cursor is referenced in subsequent Positioned UPDATE statements. If this program preparation option is not used and the UPDATE clause is not specified, the cursor cannot be referenced in a Positioned UPDATE statement. For DB2 UDB for z/OS and OS/390 the program preparation option is STDSQL(YES) or NOFOR, for DB2 UDB for UWO it is LANGLEVEL SQL92E.

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
Specifies a search condition, as described in "Search Conditions" on page 126. Each *column-name* in the *search-condition*, other than in a subquery, must name a column of the table or view. The *search-condition* must not include a subquery where the base object of both the UPDATE and the subquery is the same table.

The *search-condition* is applied to each row of the table or view and the updated rows are those for which the result of the *search-condition* is true.

If *search-condition* contains a subquery, the subquery can be thought of as being executed each time the *search-condition* is applied to a row, and the results used in applying the *search-condition*. In actuality, a subquery with no correlated references may be executed only once, whereas a subquery with a correlated reference may have to be executed once for each row.

CURRENT OF *cursor-name*
Identifies the cursor to be used in the update operation. The *cursor-name* must identify a declared cursor as explained in "DECLARE CURSOR" on page 381.

The table or view specified must also be identified in the FROM clause of the *select-statement* of the cursor, and the cursor must be updatable. For an explanation of updatable cursors, see "DECLARE CURSOR" on page 381.

When the UPDATE statement is executed, the cursor must be open and positioned on a row and that row is updated.

G In DB2 UDB for z/OS and OS/390, if a Positioned UPDATE statement is
G embedded in a program, the associated DECLARE CURSOR statement
G must include a *select-statement* rather than a *statement-name*.

isolation-clause
Specifies the isolation level used by the statement.

WITH

Introduces the isolation level, which may be one of:
- RR Repeatable read
- RS Read stability
- CS Cursor stability

If *isolation-clause* is not specified, the default isolation is used. See "Isolation Level" on page 13 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 58.

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 376.

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 return 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, changes from this statement, referential constraints, and any triggered SQL statements are rolled back.

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. The value in SQLERRD(3) does not include the number of rows that were updated as a result of a trigger. For a description of the SQLCA, see Appendix C, "SQL Communication Area (SQLCA)" on page 525.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful UPDATE statement. Until these locks are released by a commit or rollback operation, an updated row can only be accessed by:
- the application process that performed the update,
- another application process using isolation level UR through a read-only cursor, a SELECT INTO statement, or a subquery.

The locks can prevent other application processes from performing operations on the table.

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.

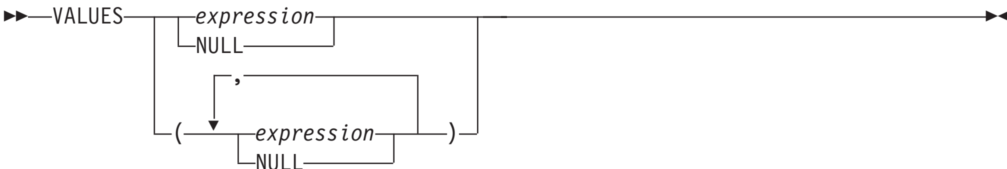
Invocation

This statement can only be used in the triggered action of a CREATE TRIGGER statement.

Authorization

None required.

Syntax



Description

VALUES
Introduces a single row consisting of one of more columns.

expression
Specifies any expression of the type described in "Expressions" on page 96.

NULL
Specifies the null value.

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.

Examples

Example: Create an AFTER trigger EMPISRT1 that invokes user-defined function NEWEMP when the trigger is activated. An insert operation on table EMPLOYEE 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
      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.

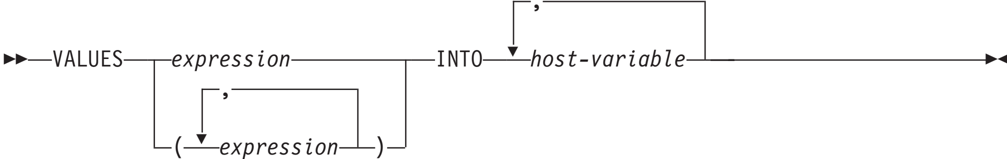
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

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 96. The expression must not include a column name. Host structures are not supported.

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 Comparisons" on page 58. 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 C, "SQL Communication Area (SQLCA)" on page 525. 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 85.

If an assignment error occurs, the values in the host variables are unpredictable.

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.[77]

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 an error is returned. See "Datetime Assignments" on page 62 for details.

Special Register Considerations: The special register CURRENT SERVER can be referenced only in a VALUES INTO statement that results in the assignment of a single host variable and not those that result in setting more than one value.

Examples

Example 1: Assign the value of the CURRENT PATH special register to host variable HV1.

```
EXEC SQL VALUES CURRENT PATH
         INTO :HV1;
```

Example 2: Assume that LOB locator LOB1 is associated with a CLOB value. Assign a portion of the CLOB value to host variable DETAILS using the LOB locator, and assign CURRENT TIMESTAMP to the host variable TIMETRACK.

```
EXEC SQL VALUES (SUBSTR(:LOB1,1,35), CURRENT TIMESTAMP)
         INTO :DETAILS, :TIMETRACK;
```

77. In DB2 UDB for UWO, the database configuration parameter dft_sqlmathwarn must be set to yes for this behavior to be supported.

WHENEVER

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

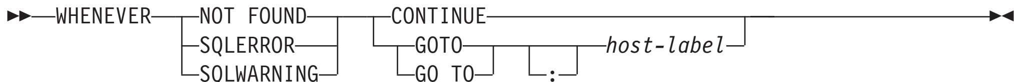
Invocation

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 "Handling SQL Errors and Warnings in Java" on page 639 or "Handling SQL Errors and Warnings in REXX" on page 647 for more information.

Authorization

None required.

Syntax



Description

The NOT FOUND, SQLERROR, or SQLWARNING clause is used to identify the type of exception condition. See Appendix E, "SQLSTATE Values—Common Return Codes" on page 539.

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 GOTO clause is used to specify the next statement to be executed when the identified type of exception condition exists.

CONTINUE
Specifies the next sequential statement of the source program.

GOTO or **GO TO** *host-label*
Specifies the statement identified by *host-label*. For host-label, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language. In a COBOL program, for example, it can be a *section-name* or an unqualified *paragraph-name*.

Notes

WHENEVER statement scope: Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type (NOT FOUND, SQLERROR, and SQLWARNING). 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.

Subroutines are supported in COBOL and C. However, normal COBOL and C scoping rules are not followed. That is, the last WHENEVER statement specified in the program source prior to the subroutine remains in effect for that subroutine. The label referenced in the WHENEVER statement must be duplicated within that subroutine. Alternatively, the subroutine could specify a new WHENEVER statement.

Examples

The following statements can be embedded in a COBOL program.

Example 1: Go to the label HANDLER for any statement that produces an error.

```
EXEC SQL  WHENEVER SQLERROR GOTO HANDLER  END-EXEC.
```

Example 2: Continue processing for any statement that produces a warning.

```
EXEC SQL  WHENEVER SQLWARNING GOTO CONTINUE  END-EXEC.
```

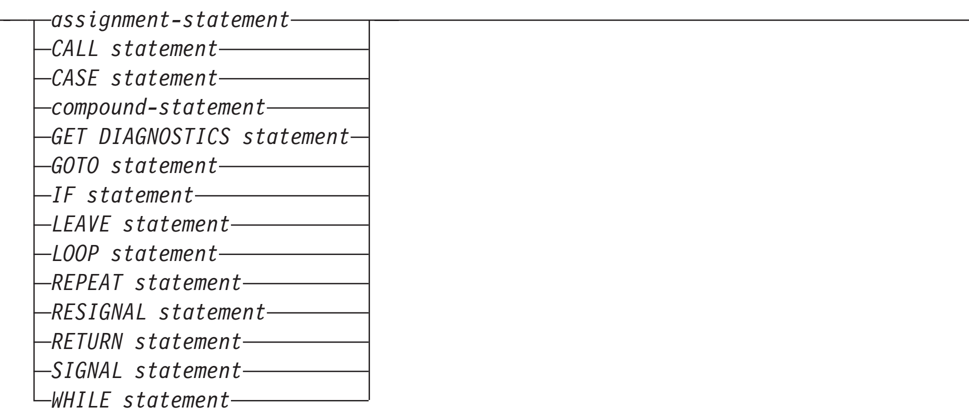
Example 3: Go to the label ENDDATA for any statement that does not return data when expected to do so.

```
EXEC SQL  WHENEVER NOT FOUND GOTO ENDDATA  END-EXEC.
```

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:



Control statements are supported in SQL procedures. SQL procedures are created by specifying LANGUAGE SQL and an SQL routine body on the CREATE PROCEDURE statement. The SQL routine body must be a single SQL statement which may be an SQL control statement.

The remainder of this chapter contains a description of the control statements 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 SQL Variables" on page 478. 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 479.

For syntax and additional information on the SQL control statements see the following topics:

- "assignment-statement" on page 480
- "CALL Statement" on page 481
- "CASE Statement" on page 482
- "compound-statement" on page 484
- "GET DIAGNOSTICS Statement" on page 491
- "GOTO Statement" on page 493
- "IF Statement" on page 494
- "LEAVE Statement" on page 496
- "LOOP Statement" on page 497

© Copyright IBM Corp. 1982, 2002

- “REPEAT Statement” on page 498
- “RESIGNAL Statement” on page 500
- “RETURN Statement” on page 502
- “SIGNAL Statement” on page 504
- “WHILE Statement” on page 506

References to SQL Parameters and SQL Variables

SQL parameters and SQL variables can be referenced anywhere in the 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. 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 or 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 or SQL parameter name. If not found, it will be assumed to be a column.

G In DB2 UDB for z/OS and OS/390, existence of tables or views is not checked
G during routine creation and therefore the name will always be checked first for an
G SQL variable or SQL parameter name.

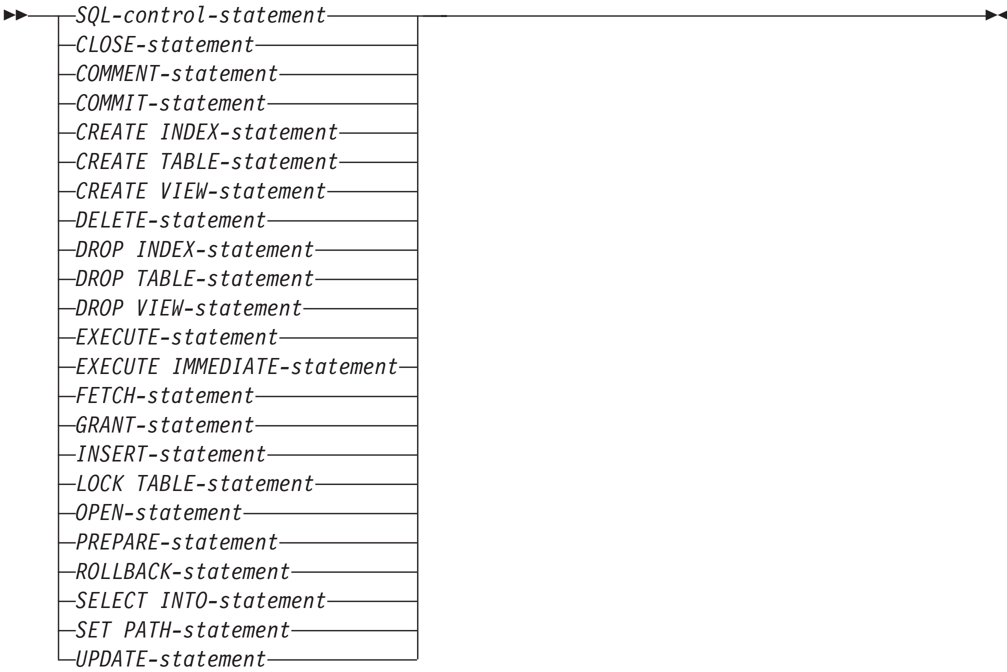
The name of an SQL parameter or SQL variable in an SQL routine can be the same as the name of an identifier used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to the identifier or to the SQL parameter or SQL variable:

- In the SET PATH statement, the name is checked as an SQL parameter or SQL variable name. If not found as an SQL variable or SQL parameter name, it will then be used as an identifier.
- In the CONNECT statement, the name is used as an identifier.

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



Notes

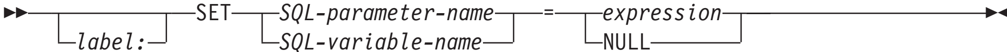
Comments: Comments can be included within the body of an SQL procedure. In addition to the double-dash form of comments (--), a comment can begin with /* and end with */. The following rules apply to this form of a comment.

- 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

The assignment statement assigns a value to an SQL parameter or an SQL variable.

Syntax



Description

label
: Specifies the label for the assignment statement. The label must be unique within the procedure and cannot be the same as the procedure name.

SQL-parameter-name
: Identifies the SQL parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE statement.

SQL-variable-name
: Identifies the SQL variable that is the assignment target. SQL variables can only be declared in a *compound-statement* and must be declared before they are used.

expression **or NULL**
: Specifies the expression or value that is the source for the assignment.

Notes

Assignment rules: Assignments in the assignment statement must conform to the SQL assignment rules as described in "Assignments and Comparisons" on page 58. If assigning to a string variable, storage assignment rules apply.

Assignments involving SQL parameters: An IN parameter can appear on the left or right side in an assignment statement. When control returns to the caller, the original value of the IN parameter is retained. An OUT parameter can also appear on the left or right side in an assignment statement. If used without first being assigned a value, the value is undefined. When control returns to the caller, the last value that is assigned to an OUT parameter is returned to the caller. For an INOUT parameter, the first value of the parameter is determined by the caller, and the last value that is assigned to the parameter is returned to the caller.

Examples

Example 1: Increase the SQL variable `p_salary` by 10 percent.

```
SET p_salary = p_salary * 1.10
```

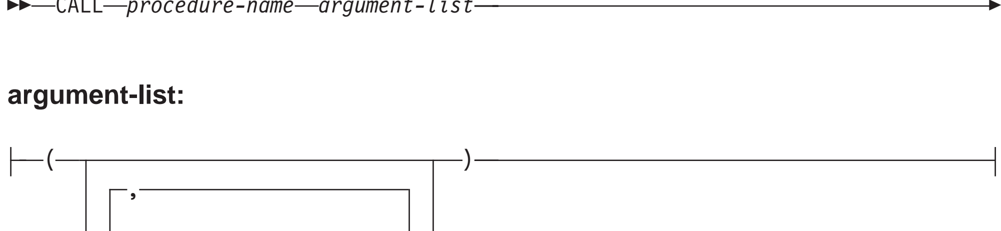
Example 2: Set SQL variable `p_salary` to the null value.

```
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 283 for details.

Syntax



Description

procedure-name
Identifies the procedure to call. The *procedure-name* must identify a procedure that exists at the current server and the procedure must be defined as an SQL procedure or a LANGUAGE C external procedure.

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.

Notes

Related information: See "CALL" on page 283 for more information.

Examples

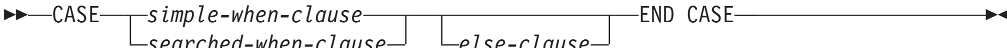
Call procedure *proc1* and pass SQL variables as parameters.

```
CALL proc1(v_empno, v_salary)
```

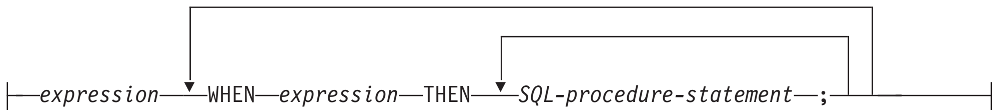
CASE Statement

The CASE statement selects an execution path based on multiple conditions.

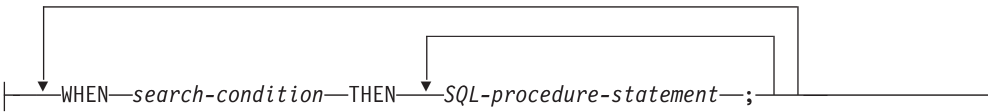
Syntax



simple-when-clause:



searched-when-clause:



else-clause:

```
|──ELSE──▼─SQL-procedure-statement──;─┘──────────────────────|
```

Description

simple-when-clause
: The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of the *expression* that follows each WHEN keyword. If the comparison is true, the statements in the associated THEN clause are executed and processing of the CASE statement ends. If the result is unknown or false, processing continues to the next comparison. If the result does not match any of the comparisons, and an ELSE clause is present, the statements in the ELSE clause are executed.

searched-when-clause
: The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are executed and processing of the CASE statement ends. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are executed.

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 returned 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 479.

Notes

Nesting 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
```

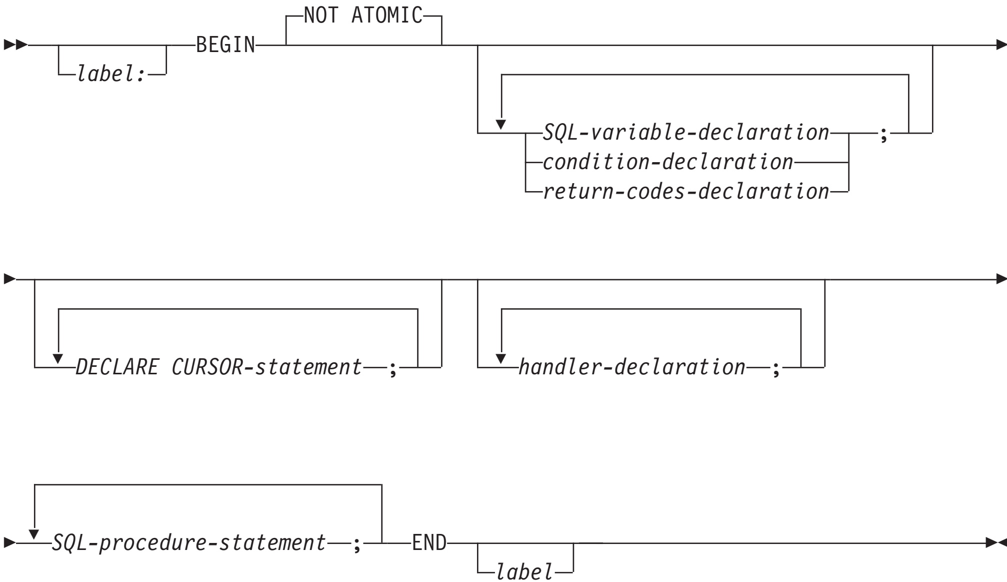
Example 2: The following example shows how to do this using the syntax for a *searched-when-clause*:

```
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
```

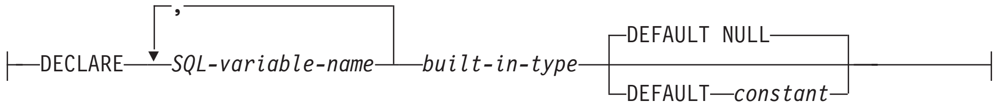
compound-statement

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:

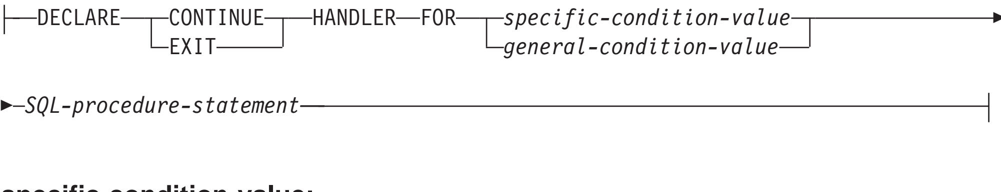


condition-declaration:

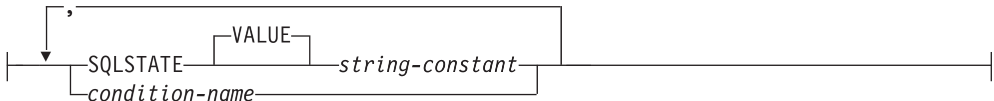
return-codes-declaration:

```
                                        ┌─DEFAULT '00000' ────────┐
├──DECLARE──┬─SQLSTATE──┬─CHARACTER(5)─┬─┼─────────────────────────┼──┬─────────┤
            │           └─CHAR(5)──────┘ └─DEFAULT──string-constant─┘  │
            │                        ┌─DEFAULT 0──────────────────┐    │
            └─SQLCODE──┬─INTEGER─┬───┼────────────────────────────┼────┘
                       └─INT─────┘   └─DEFAULT──integer-constant──┘
```

handler-declaration:



specific-condition-value:

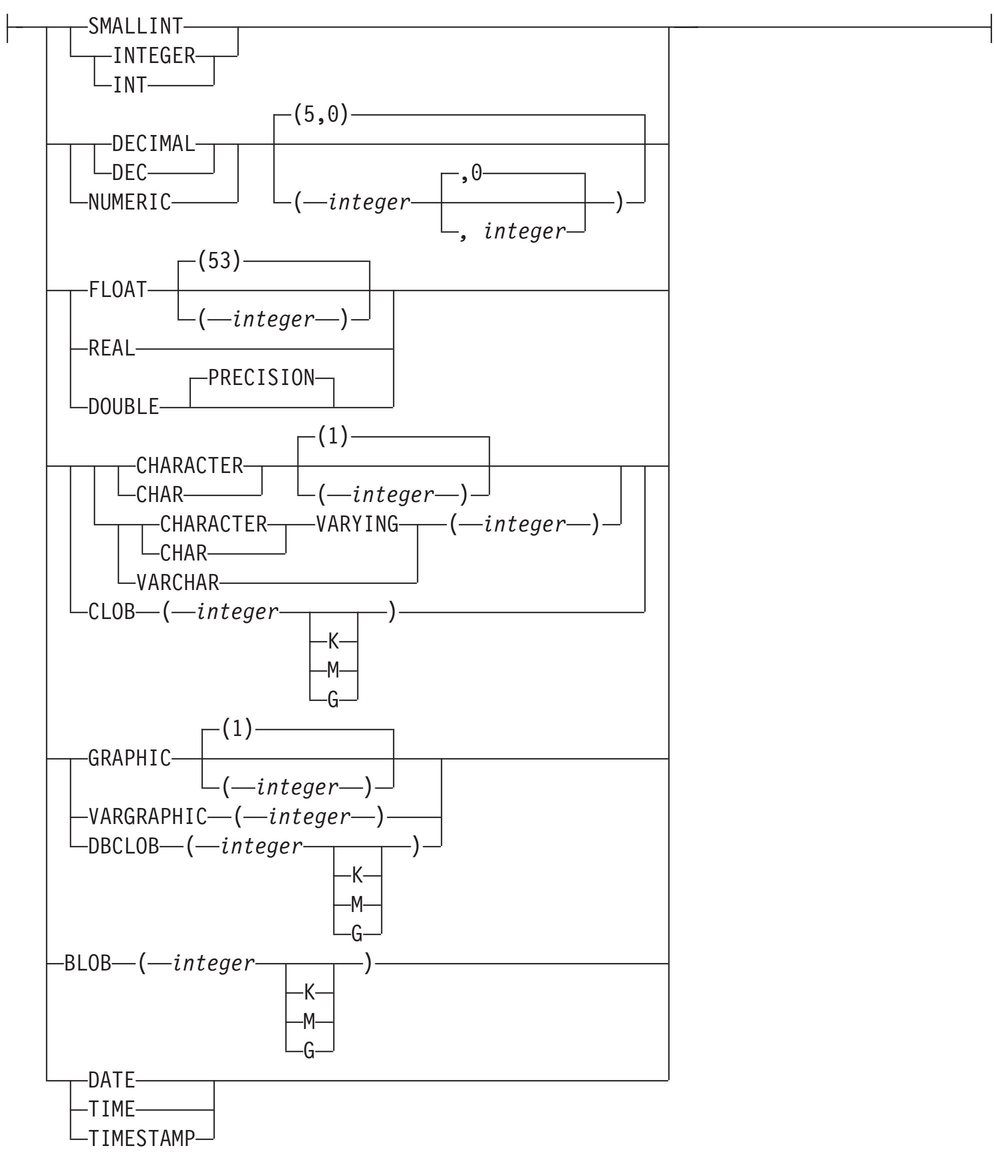


general-condition-value:

```
├──┬─SQLEXCEPTION─┬──────────────────────────────────────────┤
   ├─SQLWARNING───┤
   └─NOT FOUND────┘
```

built-in-type:

compound-statement



Description

label

Specifies the label for the *compound-statement*. If the beginning label is specified, it can be used to qualify SQL variables declared in the *compound-statement* and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label. The label must be unique within the procedure and cannot be the same as the procedure name.

NOT ATOMIC

NOT ATOMIC indicates that an unhandled error within the *compound-statement* does not cause the *compound-statement* to be rolled back.

SQL-variable-declaration

Declares an SQL variable that is local to the *compound-statement*.

SQL-variable-name

Defines the name of a local SQL variable. The database manager converts all undelimited SQL variable names to uppercase. The name must not be the same as another SQL variable within the same

compound-statement and cannot be the same as a parameter name. SQL variable names should not be the same as column names or parameter names. See "References to SQL Parameters and SQL Variables" on page 478 for how SQL variable names are resolved when there are columns with the same name involved in a statement. SQL variable names should not begin with 'SQL'.

built-in-type
Specifies the data type of the SQL variable. Refer to "Data Types" on page 42 for a description of SQL data types.

G G DB2 UDB for z/OS and OS/390 does not support LOB types for SQL variables.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The SQL variable is initialized when the SQL procedure is called. If a default value is not specified, the SQL variable is initialized to 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 within the *compound-statement* in which it is declared.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE that is associated with the condition. The *string-constant* must be specified as five characters and cannot be '00000'.

return-codes-declaration
Declares special SQL variables called SQLSTATE and SQLCODE that are set automatically to the SQL return codes returned after executing an SQL statement. Assignment to these SQL variables is not prohibited. However, assignment is ignored by handlers and processing of the next SQL statement replaces 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 is defined that handles 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 procedure body. Each cursor must have a unique name within the *compound-statement* in which it is declared. The cursor can be referenced only from within the *compound-statement*. Use an OPEN statement to open the cursor, a FETCH statement to read a row using the cursor, and a CLOSE statement to close the cursor. If the cursor is intended for use as a result set:

- 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 for the cursor 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 381.

handler-declaration
Specifies a *handler*, an *SQL-procedure-statement* to execute when an exception or completion condition occurs in the *compound-statement*. *SQL-procedure-statement* is a statement that executes when the handler receives control. A *handler-declaration* can only include a single *SQL-procedure-statement* that does not contain any other *SQL-procedure-statement*s.

A handler is active for the set of *SQL-procedure-statement*s that follow the *handler-declaration*s within the *compound-statement* in which it is declared.

There are two types of condition handlers:

CONTINUE
Specifies that after the handler is activated and completes successfully, control is returned to the SQL statement that follows the statement that returned the exception. If the error that returned the exception is an IF, CASE, WHILE, or REPEAT statement (but not an SQL-procedure-statement within one of these), then control returns to the statement that follows END IF, END CASE, END WHILE, or END REPEAT.

EXIT
Specifies that after the handler is activated and completes successfully, control is returned to the end of the *compound-statement* that declared the handler.

The condition that causes the handler to be invoked are defined in the *handler-declaration* as follows.

SQLSTATE VALUE *string*
Specifies that the handler is invoked when the specific SQLSTATE occurs. The first two characters of the SQLSTATE value must not be '00'.

condition-name
Specifies that the handler is invoked when the specific SQLSTATE associated with the condition name 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'.

If the *SQL-procedure-statement* is either a SIGNAL or RESIGNAL statement with an exception SQLSTATE, the procedure will exit with the specified exception since there is no handler in the scope of this exception.

Notes

Nesting compound statements: Compound statements cannot be nested.

Rules for handler-declarations:

- Handler declarations within the same *compound-statement* cannot contain duplicate conditions.
- A handler declaration cannot contain the same condition code or SQLSTATE value more than once, and cannot contain an SQLSTATE value and a condition name that represent the same SQLSTATE value. For a list of SQLSTATE values and more information, refer to Appendix E, "SQLSTATE Values—Common Return Codes" on page 539.
- 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* 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 returned. 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.

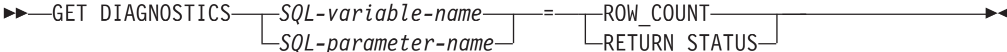
```
CREATE PROCEDURE DEPT_MEDIAN
   (IN  deptNumber   SMALLINT,
    OUT medianSalary DOUBLE)
   LANGUAGE SQL
   BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
       SELECT salary FROM staff
          WHERE DEPT = deptNumber
          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
```

GET DIAGNOSTICS Statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement that was executed.

Syntax



Description

SQL-variable-name
Identifies the SQL variable that is the assignment target. The SQL variable must be an integer variable.

SQL-parameter-name
Identifies the SQL parameter that is the assignment target. The SQL parameter must be an integer that is defined as an OUT or INOUT parameter.

ROW_COUNT
Specifies that the number of rows associated with the previous SQL statement that was executed is to be returned in the identified SQL variable or SQL parameter. 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.

RETURN_STATUS
Specifies that the status value returned from the previous SQL CALL statement is to be returned in the identified SQL variable or SQL parameter. 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 502.

G G In DB2 UDB for z/OS and OS/390, the RETURN_STATUS value is not supported.

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 the SQL procedure, these are respectively set to the SQLSTATE or SQLCODE returned from issuing the GET DIAGNOSTICS statement.

Examples

Example 1: 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
```

Example 2: Within an SQL procedure, handle the returned status value from the invocation of an SQL 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
```

GOTO Statement

The GOTO statement is used to branch to a user-defined label within an SQL routine.

Syntax

```
►►──GOTO──label─────────────────────────────────────────────────────►◄
```

Description

label
: Specifies a labelled statement where processing is to continue. appear in a *handler-declaration*.

Notes

Using a GOTO statement: It is recommended that the GOTO statement be used sparingly. This statement interferes with normal sequence of processing SQL statements, thus making a routine more difficult to read and maintain. Before using a GOTO statement, determine whether another statement, such as IF or LEAVE, can be used in place, to eliminate the need for a GOTO statement.

Examples

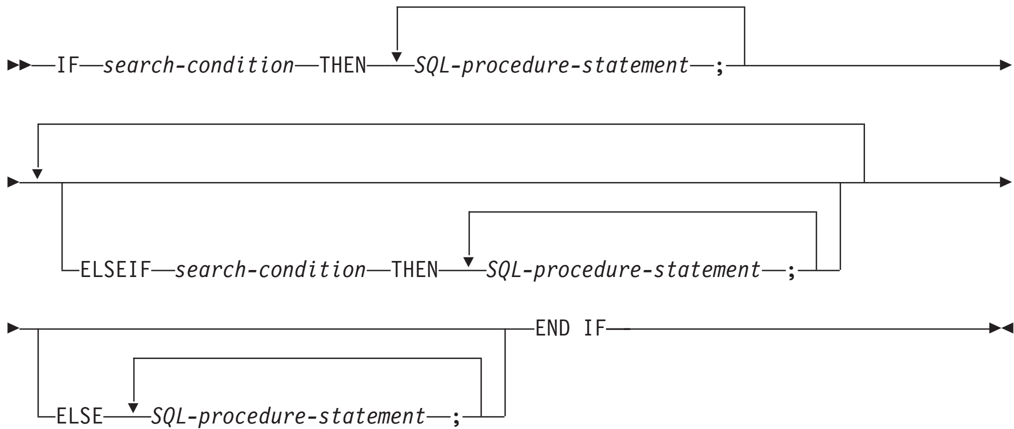
In the following compound statement used in a CREATE PROCEDURE statement, the parameters *rating* and *v_empno* are passed into the procedure, which then returns the output parameter *return_parm* as a date duration. If the employee's time in service with the company is less than 6 months, the GOTO statement 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

search-condition
: Specifies the *search-condition* for which an SQL statement should be executed. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement
: Specifies an SQL statement to be executed if the preceding *search-condition* is true.

Examples

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
   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
```

LEAVE Statement

The LEAVE statement transfers program control out of a LOOP, REPEAT, WHILE or compound statement.

Syntax

```
►►──LEAVE──label──────────────────────────────────────────────────────►◄
```

Description

label
Specifies the label of the compound, LOOP, REPEAT, or WHILE statement to exit.

Notes

Effect on open cursors: When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Examples

This example contains a loop that fetches data for cursor *c1*. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

```
CREATE PROCEDURE LEAVE_LOOP(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER;
      DECLARE v_firstnme 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 firstnme, midinit, lastname
            FROM employee;
      DECLARE CONTINUE HANDLER FOR not_found
         SET at_end = 1;
      SET v_counter = 0;
      OPEN c1;
      fetch_loop:
      LOOP
         FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
         IF at_end <> 0 THEN LEAVE fetch_loop;
         END IF;
         SET v_counter = v_counter + 1;
      END LOOP fetch_loop;
      SET counter = v_counter;
      CLOSE c1;
   END
```

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 beginning label is specified, that label can be specified on the LEAVE statement. If the ending label is specified, a matching beginning label must be specified. A label name cannot be the same as another label name in the same *compound-statement* and it cannot be the name of the SQL procedure in which the label is used.

SQL-procedure-statement
: Specifies an SQL statement to be executed in the loop.

Examples

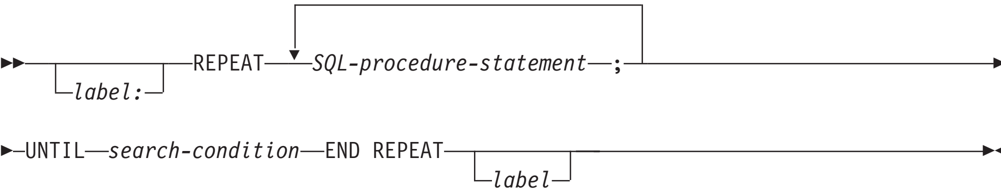
This procedure uses a LOOP statement to fetch values from the `employee` table. Each time the loop iterates, the OUT parameter *counter* is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes 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_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
            FROM employee;
      DECLARE CONTINUE HANDLER FOR NOT FOUND
         SET counter = -1;
      OPEN c1;
      fetch_loop:
      LOOP
         FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
         IF v_midinit = ' ' THEN
            LEAVE fetch_loop;
         END IF;
         SET v_counter = v_counter + 1;
      END LOOP fetch_loop;
      SET counter = v_counter;
      CLOSE c1;
   END
```

REPEAT Statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label
: Specifies the label for the REPEAT statement. If the beginning label is specified, that label can be specified on the LEAVE statements. If an ending label is specified, a matching beginning label also must be specified. A label name cannot be the same as another label name in the same *compound-statement* and it cannot be the name of the SQL procedure in which the label is used.

SQL-procedure-statement
: Specifies an SQL statement to be executed within the REPEAT loop.

search-condition
: The *search-condition* is evaluated after each execution of the REPEAT loop. If the condition is true, the loop will exit. If the condition is unknown or false, the looping continues.

Examples

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_firstnme 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 firstnme, midinit, lastname
           FROM employee;
     DECLARE CONTINUE HANDLER FOR not_found
        SET at_end = 1;
     OPEN c1;
     fetch_loop:
     REPEAT
        FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
        SET v_counter = v_counter + 1;
        UNTIL at_end > 0
```

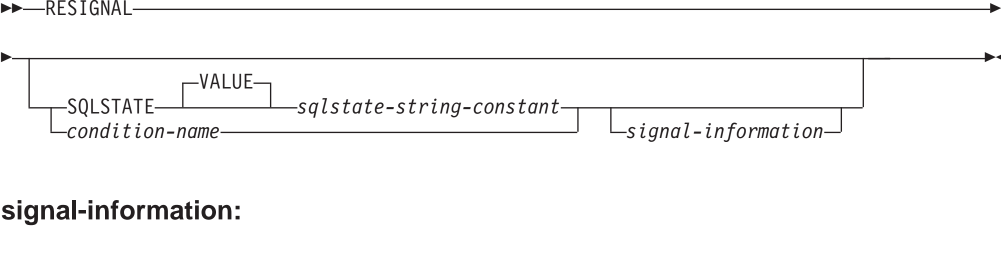
```
   END REPEAT fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

RESIGNAL Statement

The RESIGNAL statement is used within a handler to resignal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

G In DB2 UDB for z/OS and OS/390, the RESIGNAL statement is not supported.

Syntax



signal-information:

```
├──SET──MESSAGE_TEXT── = ──┬─SQL-variable-name──────────┬──────────────────────┤
                           ├─SQL-parameter-name─────────┤
                           └─diagnostic-string-constant─┘
```

Description

SQLSTATE VALUE *sqlstate-string-constant*
: Specifies the SQLSTATE 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 upper case letters ('A' through 'Z') without diacritical marks
- 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 a condition that will be returned. 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 string is longer than 70 bytes, it is truncated without 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 a CHAR or VARCHAR data type.

SQL-parameter-name
: Identifies an SQL parameter, defined for the procedure, that contains the message text. The SQL parameter must be defined as a 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 on SQLSTATEs, see Appendix E, "SQLSTATE Values—Common Return Codes" on page 539.

Processing a RESIGNAL statement:

- If the RESIGNAL statement is specified without an SQLSTATE clause or a *condition-name*, the identical condition that activated the handler is returned.
- If a RESIGNAL statement is issued, and an SQLSTATE or *condition-name* was specified, the SQLCODE returned is based on the SQLSTATE value as follows:
  – If the specified SQLSTATE class is either '01' or '02', a warning or not found is returned and the SQLCODE is set to +438
  – Otherwise, an exception is returned and the SQLCODE is set to –438.

Examples

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.

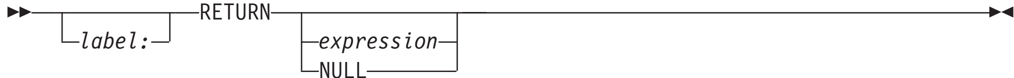
```
CREATE PROCEDURE divide
   (IN numerator INTEGER,
    IN denominator INTEGER,
    OUT divide_result INTEGER)
   LANGUAGE SQL
   CONTAINS SQL
   BEGIN
      DECLARE overflow CONDITION FOR SQLSTATE '22003';
      DECLARE CONTINUE HANDLER FOR overflow
          RESIGNAL SQLSTATE'22375' ;
      IF denominator = 0 THEN
         SIGNAL overflow;
      ELSE
          SET divide_result = numerator / denominator;
      END IF;
   END
```

RETURN Statement

The RETURN statement is used to return from the routine. For SQL functions, it returns the result of the function. For an SQL procedure, it optionally returns an integer status value.

G In DB2 UDB for z/OS and OS/390, the RETURN statement is not supported in an
G SQL procedure.

Syntax



Description

label
Specifies the label for the RETURN statement. A label can only be specified for a RETURN statement within a *compound-statement*. A label name cannot be the same as another label name in the same *compound-statement* and it cannot be the name of the SQL procedure 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 of *expression* must conform to the SQL assignment rules as described in "Assignments and Comparisons" on page 58. 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.

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.
- 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 or JDBC 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.

Examples

Example 1: 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;
   FAIL: RETURN -200;
END
```

Example 2: Define a scalar function that returns the tangent of a value using the existing sine and cosine functions.

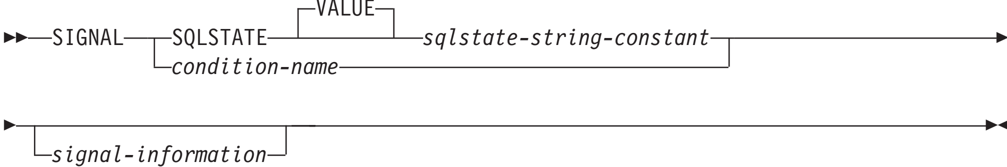
```
 CREATE FUNCTION mytan (x DOUBLE)
   RETURNS DOUBLE
   LANGUAGE SQL
   CONTAINS SQL
   NO EXTERNAL ACTION
   DETERMINISTIC
   RETURN SIN(x)/COS(x)
```

SIGNAL Statement

The SIGNAL statement is used to signal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

G In DB2 UDB for z/OS and OS/390, the SIGNAL statement is not supported.

Syntax



signal-information:

```
|──┬─SET──MESSAGE_TEXT── = ──┬─SQL-variable-name──────────┬─┬──────────|
   │                         ├─SQL-parameter-name─────────┤ │
   │                         └─diagnostic-string-constant─┘ │
   └─(──diagnostic-string-constant──)──────────────────────┘
```

Description

SQLSTATE VALUE *sqlstate-string-constant*
: Specifies an SQLSTATE 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 upper case letters ('A' through 'Z') without diacritical marks
- 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*.

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 string is longer than 70 bytes, it is truncated without 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 a CHAR or VARCHAR data type.

SQL-parameter-name
: Identifies an SQL parameter, defined for the procedure, that contains the message text. The SQL parameter must be defined as a 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);
```

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 on SQLSTATEs, see Appendix E, “SQLSTATE Values—Common Return Codes” on page 539.

Processing a SIGNAL statement: If a SIGNAL statement is issued, the SQLCODE returned is based on the SQLSTATE value as follows:

- If the specified SQLSTATE class is either ’01’ or ’02’, a warning or not found is returned and the SQLCODE is set to +438
- Otherwise, an exception is returned and the SQLCODE is set to −438.

Examples

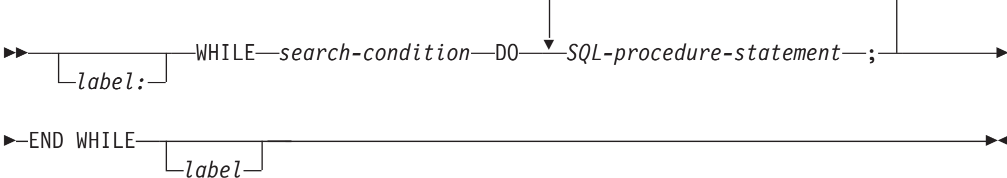
An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
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 or group of statements while a specified condition is true.

Syntax



Description

label
: Specifies the label for the WHILE statement. If the beginning label is specified, it can be specified in the LEAVE statement. If the ending label is specified, it must be the same as the beginning label. A label name cannot be the same as another label name in the same *compound-statement* and it cannot be the name of the SQL procedure in which the label is used.

search-condition
: Specifies a condition that is evaluated before each execution of the 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.

Examples

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
            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 SQL and database limits imposed by the IBM relational database products. Adhering to the most restrictive case can help the programmer design application programs that are easily portable.

Note:

- System storage limits may preclude the limits specified here. For example, see “Byte Counts” on page 366.
- A limit of *storage* means that the limit is dependent on the amount of storage available.
- A limit of *statement* means that the limit is dependent on the limit for the maximum length of a statement.

Table 37. Identifier Length Limits

Identifier Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Longest authorization name	8	10	30	8
Longest condition name	64	128	64	64
Longest constraint name	8[82]	128	18	8[82]
Longest correlation name	18	128	128	18
Longest cursor name	18	18	18	18
Longest external program name (unqualified form)	8	10	18	8
Longest external program name (string form)	1305	279	254	254
Longest host identifier[79]	64	64	255	64
Longest schema name	8[78]	10	8[80]	8
Longest server name	16	18	8	18
Longest statement name	18	18	18	18
Longest SQL label	64	128	64	64
Longest unqualified alias name	18	128	128	18
Longest unqualified column name	18	30	30	18
Longest unqualified distinct type name	18	128	18	18
Longest unqualified function name	18	128	18	18
Longest unqualified index name	18	128	18	18
Longest unqualified package name	8	10	8	8
Longest unqualified procedure name	18	128	128	18
Longest unqualified specific name	18	128	128	18
Longest unqualified SQL parameter name	18	128	64[81]	18
Longest unqualified SQL variable name	18	128	64	18

© Copyright IBM Corp. 1982, 2002

Table 37. Identifier Length Limits (continued)

Identifier Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Longest unqualified table and view name	18	128	128	18
Longest unqualified trigger name	8	128	18	18

Table 38. Numeric Limits

Numeric Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Smallest SMALLINT value	−32 768	−32 768	−32 768	−32 768
Largest SMALLINT value	+32 767	+32 767	+32 767	+32 767
Smallest INTEGER value	−2 147 483 648	−2 147 483 648	−2 147 483 648	−2 147 483 648
Largest INTEGER value	+2 147 483 647	+2 147 483 647	+2 147 483 647	+2 147 483 647
Largest decimal precision	31	31	31	31
Smallest DOUBLE value[83]	$-7.2x10^{75}$	$-1.79x10^{308}$	$-1.79x10^{308}$	$-7.2x10^{75}$
Largest DOUBLE value[83]	$+7.2x10^{75}$	$+1.79x10^{308}$	$+1.79x10^{308}$	$+7.2x10^{75}$
Smallest positive DOUBLE value[83]	$+5.4x10^{-79}$	$+2.23x10^{-308}$	$+2.23x10^{-307}$	$+5.4x10^{-79}$
Largest negative DOUBLE value[83]	$-5.4x10^{-79}$	$-2.23x10^{-308}$	$-2.23x10^{-307}$	$-5.4x10^{-79}$
Smallest REAL value[83]	$-7.2x10^{75}$	$-3.4x10^{38}$	$-3.4x10^{38}$	$-3.4x10^{38}$
Largest REAL value[83]	$+7.2x10^{75}$	$+3.4x10^{38}$	$+3.4x10^{38}$	$+3.4x10^{38}$
Smallest positive REAL value[83]	$+5.4x10^{-79}$	$+1.18x10^{-38}$	$+1.17x10^{-37}$	$+1.17x10^{-37}$
Largest negative REAL value[83]	$-5.4x10^{-79}$	$-1.18x10^{-38}$	$-1.17x10^{-37}$	$-1.17x10^{-37}$

Table 39. String Limits

String Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Maximum length of CHAR (in bytes)	255	32 765[85]	254	254
Maximum length of VARCHAR (in bytes)	32 704	32 739[85]	32 672	32 672
Maximum length of CLOB (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of GRAPHIC (in double-byte characters)	127	16 382[85]	127	127
Maximum length of VARGRAPHIC (in double-byte characters)	16 352	16 369[85]	16 336	16 336

78. 18 for packages.

79. Individual host languages may vary.

80. The schema name can be up to 30 bytes for the schema name of all objects except distinct types.

81. The limit is 128 for external routines.

82. For referential constraints. The limit for other constraints is 18.

83. The values shown are approximate.

Table 39. String Limits (continued)

String Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Maximum length of DBCLOB (in double-byte characters)	1 073 741 823	1 073 741 823	1 073 741 823	1 073 741 823
Maximum length of BLOB (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Maximum length of character constant	255	32 740	32 672	255
Maximum length of a graphic constant[84]	124	16 370	16 336	124
Maximum length of a concatenated character string	2 147 483 647	2 147 483 647	2 147 483 647	32 766
Maximum length of a concatenated graphic string	1 073 741 823	1 073 741 823	1 073 741 823	16 370
Maximum length of a concatenated binary string	2 147 483 647	2 147 483 647	2 147 483 647	32 766
Maximum number of hex constant digits	254	65 480	16 336	254
Maximum length of catalog comments (in bytes)	254	2000	254	254

Table 40. Datetime Limits for IBM SQL and All IBM Relational Database Products

Datetime Limits[86]	DB2 UDB SQL and All IBM Relational Database Products
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 41. Database Manager Limits

Database Manager Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Most columns in a table	749[89]	8000	1012	750
Most columns in a view	750	8000	5000	750
Maximum length of a row including all overhead	32 714[88]	32 766	32 677[88]	32 677
Maximum number of parameters in a function	200[97]	90	90	90

84. Further restricted by individual utilities and preprocessors.

85. If the column is NOT NULL, the limit is one more.

86. Shown in ISO format.

Table 41. Database Manager Limits (continued)

Database Manager Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Maximum number of parameters in a procedure	200[98, 97]	253[96]	90	90
Maximum size of a table[90, 87]	64 gigabytes	1 terabyte	512 gigabytes	512 gigabytes
Maximum size of an index[90, 87]	64 gigabytes	1 terabyte	512 gigabytes	512 gigabytes
Maximum number of rows in a table[83, 87]	$1x10^{12}$	4 294 967 288	$4x10^9$	$4x10^9$
Longest index key[91]	255[92]	2000	1024	255
Most columns in an index key	64	120	16	16
Most indexes on a table	storage	4000[83]	32 767 or storage	4000[83]
Most tables referenced in an SQL statement	225[93]	256	storage	225[93]
Most tables referenced in a view	225[93]	32	storage	32
Most host variable declarations in a precompiled program	storage	storage	storage	storage
Most host variables in an SQL statement	statement	statement	4096	4096
Longest host variable value used for insert or update (in bytes)	2 147 483 647	2 147 483 647	2 147 483 647	2 147 483 647
Longest SQL statement (in bytes)	32 765	32 767	65 535	32 765
Longest CHECK constraint (in bytes)	3800	statement	65 535	3800
Most elements in a select list	750	8000	1012	750
Most predicates in a WHERE or HAVING clause	750	statement	statement	750
Maximum number of columns in a GROUP BY clause	4000	120	1012	120
Maximum total length of columns in a GROUP BY clause	4000	2000	32 677	2000
Maximum number of columns in an ORDER BY clause	4000	10 000	1012	1012
Maximum total length of columns in an ORDER BY clause	4000	10 000	32 677	4000
Maximum number of prepared statements	storage	storage	storage[94]	storage
Most declared cursors in a program	storage	storage	storage	storage
Maximum number of cursors opened at one time	storage	storage	storage	storage
Most tables in a relational database[90]	storage	storage	storage	storage
Maximum number of triggers on a table	storage	300	storage	300
Maximum number of nested trigger levels	16[95]	200	16	16
Maximum length of a password	8	128	18	8
Maximum number of constraints on a table	storage	300	storage	300

Table 41. Database Manager Limits (continued)

Database Manager Limits	DB2 UDB for z/OS and OS/390	DB2 UDB for iSeries	DB2 UDB for UWO	DB2 UDB SQL
Maximum length of a path	254	3483	254	254

87. The limits are greater when partitioned. See product documentation.
88. Row size may be further restricted by the page size of the table space.
89. If the table is not a dependent table, the limit is 750.
90. The numbers shown are architectural limits and approximations. The practical limits may be less.
91. The longest index key is actually the number provided in the table minus the number of columns that allow nulls.
92. The maximum can be less depending on index options.
93. This is an approximate guideline. In a complex SELECT, the number of tables that can be joined may be significantly less.
94. In REXX, the maximum number of prepared statements is 100. Of these, no more than 50 can be declared cursors with the WITH HOLD clause, and no more than 50 can be declared cursors without the WITH HOLD clause.
95. Further limited by the presence of nested procedures and functions.
96. Depending on parameter style, the limit may be one or two more.
97. For some languages the limit may be more.
98. The limit may be less in the presence of result sets.

Appendix B. Characteristics of SQL Statements

This appendix contains information on the characteristics of SQL statements pertaining to the various places where they are used.

- "Actions Allowed on SQL Statements" on page 516 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 Classification for Routines" on page 518 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 520 provides information about the use of SQL statements when the application server is not the same as the application requestor.

© Copyright IBM Corp. 1982, 2002

Actions Allowed on SQL Statements

Table 42 shows whether a specific SQL statement can be executed, issued interactively or prepared dynamically, or processed by the requester, the server, or the precompiler. The letter **Y** means *yes*.

Table 42. Actions allowed on SQL statements

SQL statement	Executable	Issued interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
ALTER	Y	Y		Y	
BEGIN DECLARE SECTION[100,101]					Y
CALL[99]	Y			Y	
CLOSE[100]	Y			Y	
COMMENT	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (Type 1 and Type 2)[100,101]	Y		Y		
CREATE	Y	Y		Y	
DECLARE CURSOR[100]					Y
DELETE	Y	Y		Y	
DESCRIBE[100]	Y			Y	
DROP	Y	Y		Y	
END DECLARE SECTION[100,101]					Y
EXECUTE[100]	Y			Y	
EXECUTE IMMEDIATE[100]	Y			Y	
FETCH	Y			Y	
FREE LOCATOR[100,101]	Y			Y	
GRANT	Y	Y		Y	
INCLUDE[100,101]					Y
INSERT	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
OPEN[100]	Y			Y	
PREPARE[100]	Y			Y	
RELEASE connection[100,101]	Y		Y		
RENAME	Y	Y		Y	
REVOKE	Y	Y		Y	
ROLLBACK	Y	Y		Y	

99. The statement can be dynamically prepared, but only from a CLI, ODBC or JDBC driver that supports dynamic CALL statements.

100. This statement is not applicable in a Java program.

101. This statement is not supported in a REXX program.

Table 42. Actions allowed on SQL statements (continued)

SQL statement	Executable	Issued interactively or dynamically prepared	Processed by		
			Requesting system	Server	Precompiler
SELECT INTO[101]	Y			Y	
SET CONNECTION[100,101]	Y		Y		
SET PATH	Y	Y		Y	
SET transition-variable[102]	Y			Y	
SQL-control-statement	Y			Y	
UPDATE	Y	Y		Y	
VALUES[102]	Y			Y	
VALUES INTO[101]	Y			Y	
WHENEVER[100,101]					Y

102. This statement can only be used in the triggered action of a trigger.

SQL Statement Data Access Classification for Routines

Table 43 indicates (using the letter Y) whether an SQL statement (specified in the first column) is allowed to execute in a routine with the specified SQL data access classification.

Table 43. SQL Data Access Classification of SQL Statements

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALTER				Y
BEGIN DECLARE SECTION	Y[103]	Y	Y	Y
CALL		Y[104]	Y[104]	Y[104]
CLOSE			Y	Y
COMMENT				Y
COMMIT		Y[106]	Y[106]	Y
CONNECT(Type 1 and Type 2)[105]				
CREATE				Y
DECLARE CURSOR	Y[103]	Y	Y	Y
DELETE				Y
DESCRIBE			Y	Y
DROP				Y
END DECLARE SECTION	Y[103]	Y	Y	Y
EXECUTE		Y[107]	Y[107]	Y
EXECUTE IMMEDIATE		Y[107]	Y[107]	Y
FETCH			Y	Y
FREE LOCATOR		Y	Y	Y
GRANT				Y
INCLUDE	Y[103]	Y	Y	Y
INSERT				Y
LOCK TABLE		Y	Y	Y
OPEN			Y	Y
PREPARE		Y	Y	Y
RELEASE connection[105]				
RENAME				Y
REVOKE				Y
ROLLBACK′		Y[106]	Y[106]	Y
SELECT INTO			Y	Y
SET CONNECTION[105]				
SET PATH		Y	Y	Y

103. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.

104. A CALL statement can only be used in a procedure defined as LANGUAGE SQL or LANGUAGE C.

105. Connection management statements are not allowed in any procedure execution contexts.

106. For DB2 UDB for UWO, an SQL procedure only allows COMMIT and ROLLBACK statements with MODIFIES SQL DATA.

Table 43. SQL Data Access Classification of SQL Statements (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
SET transition-variable				
SQL-control-statement		Y	Y	Y
UPDATE				Y
VALUES				
VALUES INTO			Y	Y
WHENEVER	Y[103]	Y	Y	Y

107. 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.

Considerations for Using Distributed Relational Database

This section contains information that may be useful in developing applications that use application servers which are not the same product as their application requesters.

All DB2 Universal Database products support extensions to the SQL described in this publication. Some of these extensions are product-specific, but many are already supported 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 via 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 requestor:

- for DB2 UDB for z/OS and OS/390 application requestor, see Table 44 on page 521
- for DB2 UDB for iSeries application requestor, seeTable 45 on page 522
- for DB2 UDB for UWO application requestor, see Table 46 on page 523.

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 may mean that the function is available only if server and requester are the same product or that the statement is blocked by the application requestor from being processed at the application server.

Table 44. DB2 UDB for z/OS and OS/390 Application Requester

SQL Statement or Function	DB2 UDB for z/OS and OS/390 Application Server	DB2 UDB for iSeries Application Server	DB2 UDB for UWO Application Server
COMMIT HOLD	R	R	R
DECLARE STATEMENT			
DECLARE TABLE			
DECLARE VARIABLE			
DESCRIBE TABLE			R
DESCRIBE with USING clause			R
DISCONNECT	R	R	R
Large Object (LOB) Data Types			R
BIGINT Data Types	R	[109]	[109]
ROWID Data Types		R	R
DATALINK Data Types	R	R	R
Distinct Data Types			[110]
Host declarations not documented in language specific appendices		[108]	[108]
PREPARE with USING clause			R
ROLLBACK HOLD	R	R	R
SET CURRENT PACKAGESET			
SET host variable		R	R
SET TRANSACTION	R	R	R
Scrollable Cursor statements	R	R	R
UPDATE cursor - FOR UPDATE OF clause not specified			

108. The statement is supported if the application requester understands it.

109. The DB2 UDB for z/OS and OS/390 application requestor will process a BIGINT data type at the application server using the compatible DECIMAL(19,0) data type.

110. The DB2 UDB for UWO application server returns the source type of the distinct type but the distinct type name is not returned.

Table 45. DB2 UDB for iSeries Application Requester

SQL Statement or Function	DB2 UDB for z/OS and OS/390 Application Server	DB2 UDB for iSeries Application Server	DB2 UDB for UWO Application Server
COMMIT HOLD	R		R
DECLARE STATEMENT			
DECLARE TABLE			
DECLARE VARIABLE			
DESCRIBE TABLE			R
DESCRIBE with USING clause			R
DISCONNECT			
Large Object (LOB) Data Types			R
BIGINT Data Types	R		
ROWID Data Types	[111]	R	R
DATALINK Data Types	R		R
Distinct Data Types			[110]
Host declarations not documented in language specific appendices	[108]		[108]
PREPARE with USING clause			R
ROLLBACK HOLD	R		R
SET CURRENT PACKAGESET	R	R	R
SET host variable	R		R
SET TRANSACTION	R		R
Scrollable Cursor statements	R		R
UPDATE cursor - FOR UPDATE OF clause not specified	R		

111. The DB2 UDB for iSeries application requestor will process a ROWID data type at the application server using the compatible VARCHAR(40) FOR BIT DATA data type.

Table 46. DB2 UDB for UWO Application Requester

SQL Statement or Function	DB2 UDB for z/OS and OS/390 Application Server	DB2 UDB for iSeries Application Server	DB2 UDB for UWO Application Server
COMMIT HOLD	R	R	R
DECLARE STATEMENT	R	R	R
DECLARE TABLE	R	R	R
DECLARE VARIABLE	R	R	R
DESCRIBE TABLE	R	R	R
DESCRIBE with USING clause	R	R	R
DISCONNECT			
Large Object (LOB) Data Types			
BIGINT Data Types	R		
ROWID Data Types	[112]	R	R
DATALINK Data Types	R	R	R
Distinct Data Types			
Host declarations not documented in language specific appendices	[108]	[108]	
PREPARE with USING clause	R	R	R
ROLLBACK HOLD	R	R	R
SET CURRENT PACKAGESET			
SET host variable	R	R	R
SET TRANSACTION	R	R	R
Scrollable Cursor statements	R	R	R
UPDATE cursor - FOR UPDATE OF clause not specified	R		

112. The DB2 UDB for UWO application requestor will process a ROWID data type at the application server using the compatible VARCHAR(40) FOR BIT DATA data type.

Appendix C. SQL Communication Area (SQLCA)

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 SQLSTATE or a stand-alone SQLCODE 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 and REXX. For information on the use of the SQLCA in a REXX program, see Appendix K, "Coding SQL Statements in REXX Applications". For information on how to access the information regarding errors and warnings in Java, see Appendix J, "Coding SQL Statements in Java Applications" on page 627.

In COBOL and C, the name of the storage area must be SQLCA. Every SQL statement must be within the scope of its declaration.

If stand-alone SQLCODE or SQLSTATE is used, an SQLCA cannot be included. For more information, see "SQL Return Codes" on page 264.

The stand-alone SQLCODE and stand-alone SQLSTATE must not be specified in Java or REXX.

Field Descriptions

Table 47. Field Descriptions for an SQLCA

C Name[113]	COBOL Name	Field Data Type	Field Value
sqlcaid	SQLCAID	CHAR(8)	Contains an 'eye catcher' for storage dumps, 'SQLCA'.
sqlcabc	SQLCABC	INTEGER	Contains the length of the SQLCA, 136.
sqlcode	SQLCODE	INTEGER	Contains an SQL return code: **0** Successful execution, although SQLWARN indicators (see below) might have been set. **positive** Successful execution, but with a warning condition. **negative** Error condition.
sqlerrml[114]	SQLERRML	SMALLINT	Contains the length for SQLERRMC, in the range 0 through 70. If the length is 0, the value of SQLERRMC is not pertinent.
sqlerrmc[114]	SQLERRMC	VARCHAR (70)	Contains information that is substituted for variables in the descriptions of error conditions. See the product references for further information.

113. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

114. In C and COBOL, SQLERRM includes SQLERRML and SQLERRMC.

© Copyright IBM Corp. 1982, 2002

Table 47. Field Descriptions for an SQLCA (continued)

C Name[113]	COBOL Name	Field Data Type	Field Value
sqlerrp	SQLERRP	CHAR(8)	Begins with a three-letter identifier indicating the product: DSN for DB2 UDB for z/OS and OS/390 QSQ for DB2 UDB for iSeries SQL for DB2 UDB for UWO If the SQLCODE indicates an error condition, then this field contains the name of the module that returned the error. See "CONNECT (Type 1)" on page 296 for additional information.
sqlerrd	SQLERRD	Array	Contains six INTEGER variables that provide diagnostic information. The third SQLERRD variable shows the number of rows affected after INSERT, UPDATE, and DELETE. If a PREPARE statement is successful, the fourth SQLERRD variable contains a relative cost estimate of the resources required to process the prepared statement. The fifth SQLERRD variable shows the number of rows affected by referential constraints as a result of a delete operation. G G In DB2 UDB for z/OS and OS/390, the use of SQLERRD(5) is not supported.
sqlwarn	SQLWARN	Array	Contains a set of warning indicators. Each indicator is either blank or contains a value as indicated below.
SQLWARN0	SQLWARN0	CHAR(1)	Contains 'W' if at least one other indicator contains 'W'; it is blank if all the other indicators are blank.
SQLWARN1	SQLWARN1	CHAR(1)	Contains 'W' if the value of a string column was truncated when assigned to a host variable.
SQLWARN2	SQLWARN2	CHAR(1)	Contains 'W' if null values were eliminated from the argument of a column function; not necessarily set to 'W' for the MIN function because its results are not dependent on the elimination of null values.
SQLWARN3	SQLWARN3	CHAR(1)	Contains 'W' if the number of columns is larger than the number of host variables.
SQLWARN4	SQLWARN4	CHAR(1)	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
G SQLWARN5	SQLWARN5	CHAR(1)	Contents are product-specific.
SQLWARN6	SQLWARN6	CHAR(1)	Contains 'W' if date arithmetic results in an end of month adjustment. For more information, see "Incrementing and decrementing dates" on page 102.
G SQLWARN7	SQLWARN7	CHAR(1)	Contents are product-specific.
SQLWARN8	SQLWARN8	CHAR(1)	Contains 'W' if a character that could not be converted was replaced with a substitution character.
G SQLWARN9	SQLWARN9	CHAR(1)	Contents are product-specific.
G SQLWARNA	SQLWARNA	CHAR(1)	Contents are product-specific.

Table 47. Field Descriptions for an SQLCA (continued)

C Name[113]	COBOL Name	Field Data Type	Field Value
sqlstate	SQLSTATE	CHAR(5)	A return code as described in Appendix E, "SQLSTATE Values—Common Return Codes" on page 539. that indicates the outcome of the most recently executed SQL statement.

INCLUDE SQLCA Declarations

For C

In C, INCLUDE SQLCA declarations are equivalent (but not necessarily identical) 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;
```

Figure 9. INCLUDE SQLCA Declarations for C

For COBOL

In COBOL, INCLUDE SQLCA declarations are equivalent (but not necessarily identical) 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).
```

Figure 10. INCLUDE SQLCA Declarations for COBOL

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, FETCH, CALL, and EXECUTE statements. An SQLDA can be used in a DESCRIBE or PREPARE statement, altered with the addresses of storage areas[115], and then reused in a FETCH statement. It can also be used in OPEN, EXECUTE or CALL statements to provide input values or output variables.

SQLDAs are supported, with predefined declarations, for C, COBOL and REXX. In REXX, the SQLDA is somewhat different than in the other languages; for information on the use of SQLDAs in REXX, see "Defining SQL Descriptor Areas in REXX" on page 644.

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 desribes either LOBs or distinct types the base SQLVARs are followed by the same number of occurrences of an *extended SQLVAR*.

Base SQLVAR
: The base SQLVAR entry is always present in an SQLDA. The fields of the base SQLVAR entry contain information about the column or storage area including data type, length attribute (except for LOBs), column name, CCSID, storage area address for data, and storage area address for an indicator.

Extended SQLVAR
: The extended SQLVAR entry is used (for each column or variable) if the SQLDA includes any LOBs or distinct types. Each extended SQLVAR entry provides extended information for the corresponding base SQLVAR entry.

---

115. A storage area could be the storage for a variable defined in the program (that may also be a host variable) or an area of storage explicitly allocated by the application.

© Copyright IBM Corp. 1982, 2002

For distinct types, the extended SQLVAR contains the distinct type name. For LOBs, the extended SQLVAR contains the length attribute of the storage area and a pointer to the storage area that contains the actual length. If locators are used to represent LOBs, an extended SQLVAR is not necessary. If the corresponding base SQLVAR represents neither a LOB or distinct type, the extended SQLVAR includes no additional information.

Field Descriptions in an SQLDA Header

Table 48. Field Descriptions for an SQLDA Header

C Name[116, 117] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the application 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 Occurrences of SQLVAR Entries are Needed" on page 531.	A '2' in the 7th byte indicates that two SQLVAR entries were allocated for each column. If the SQLNAME field contains an overriding CCSID, the 6th byte must be set to a '+' character.
sqldabc SQLDABC	INTEGER	Number of bytes of storage for 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+SQLN*(N), where N is the length of an SQLVAR occurrence.[118]
sqln SQLN	SMALLINT	Before invoking DESCRIBE or PREPARE, set to the total number of occurrences of SQLVAR entries allocated for the SQLDA. The value is not changed by the database manager during DESCRIBE or PREPARE.	Total number of occurrences of SQLVAR entries allocated in the SQLDA. SQLN must be set to a value greater than or equal to SQLD. If LOBs types are included, extended SQLVARs are required. SQLN must be set to two times the number of parameter markers in the statement.
sqld SQLD	SMALLINT	The number of columns in the result table of the select-statement. Zero if the statement being described is not a select-statement.	Number of occurrences of SQLVAR entries in the SQLDA that are used when executing the statement. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

116. The field names shown are those present in an SQLDA that is obtained via an INCLUDE statement.

117. In this column, the lowercase name is the "C Name." The uppercase name is the "COBOL Name."

118. The value of N varies depending on the environment. For portability, the value should be calculated using an appropriate language sizing function. For example, in C use the sizeof() function to determine the size of the SQLVAR.

Determining How Many Occurrences of SQLVAR Entries 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 Table 49 for more information.

If more than 1 set of SQLVARs is needed, the 7th byte of SQLDAID is 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 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 if enough SQLVARs were not specified for even the base SQLVAR entries. No SQLVAR entries are returned.

Table 49 shows how to map the base and extended SQLVAR entries. For an SQLDA that contains both base and extended SQLVAR entries, the base SQLVAR entries are in the first block, followed by a block of extended SQLVAR entries. 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 49. Contents of SQLVAR Arrays

LOBs	DISTINCT types	7th byte of SQLDAID	SQLN Minimum	First Set (Base)	Second Set (Extended)
No	No	Blank	n	Data type information	Not used
Yes	No	2	2n	Data type information with no length for LOB entries	LOB length for LOB entries
No	Yes	2	2n	Data type information except source data type information for distinct type entries	distinct type name
Yes	Yes	2	2n	Data type information with no length for LOB entries and source data type for distinct type entries	LOBs length for LOB entries and distinct type name for distinct type entries

Field Descriptions in an Occurrence of SQLVAR

Table 50. Field Descriptions for a Base SQLVAR

C Name[116, 117] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
sqltype SQLTYPE	SMALLINT	The data type of the column and whether it can contain nulls. For a description of the type codes, see Table 52 on page 534. 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.	The data type of the host variable and whether an indicator variable is provided. For a description of the type codes, see Table 52 on page 534.
sqllen SQLLEN	SMALLINT	The length attribute of the column. For datetime columns, the length of the string representation of the values. See Table 52 on page 534. 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 52 on page 534. 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.
sqldata SQLDATA	pointer[119]	For string columns, the CCSID of the column. For datetime columns, SQLDATA can contain the CCSID of the string representation of the datetime value. See Table 53 on page 535 for the format of the field.	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).
G sqlname SQLNAME	VARCHAR (30)	The unqualified name of the column. If the column does not have a name, the contents are product-specific. The name is case sensitive and does not contain surrounding delimiters.	For SQLVARs representing string types, the CCSID of the string. See Table 53 on page 535 for the format of the field.

119. There may be additional reserved bytes preceding this field to properly align the pointer. See each product's SQLDA include file for details.

120. There are additional reserved bytes preceding this field to properly align the pointer and make the structure the same size as the base SQLVAR. See each product's SQLDA include file for details.

Table 51. Field Descriptions for an Extended SQLVAR

C Name[116, 117] COBOL Name	Field Data Type	Usage in DESCRIBE or PREPARE (set by the database manager)	Usage in FETCH, OPEN, CALL, or EXECUTE (set by the user prior to executing the statement)
len.sqllonglen SQLLONGLEN	INTEGER	The length attribute of a LOB column.	The length attribute of a LOB host variable. The length attribute indicates the number of bytes for a BLOB or CLOB, and the number of double-byte characters for a DBCLOB. The database manager ignores the SQLLEN field in the base SQLVAR for these data types.
sqldatalen SQLDATALEN	pointer[120]	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 SQLDATATYPE-NAME	VARCHAR (30)	The fully qualified distinct type name for a distinct type column.	Not used.

SQLTYPE and SQLLEN

The following table shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In an SQLDA used in DESCRIBE or PREPARE statements, 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 null value.

In an SQLDA used in FETCH, OPEN, or EXECUTE statements, 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 52. SQLTYPE and SQLLEN values

	For DESCRIBE and PREPARE		For FETCH, OPEN, CALL, and EXECUTE	
SQLTYPE	**Column Data Type**	**SQLLEN**	**Host Variable Data Type**	**SQLLEN**
384/385	date	10	fixed-length character-string representation of a date	length attribute of the host variable
388/389	time	8	fixed-length character-string representation of a time	length attribute of the host variable
392/393	timestamp	26	fixed-length character string representation of a timestamp	length attribute of the host variable
400/401	Not Applicable	Not Applicable	NUL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 [121]	BLOB	Not used. [121]
408/409	CLOB	0 [121]	CLOB	Not used. [121]
412/413	DBCLOB	0 [121]	DBCLOB	Not used. [121]
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
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 [122]	precision in byte 1; scale in byte 2	zoned decimal [122]	precision in byte 1; scale in byte 2
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
504/505	Not Applicable	Not Applicable	DISPLAY SIGN LEADING SEPARATE [123]	precision in byte 1; scale in byte 2
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

121. Field SQLLONGLEN in the extended SQLVAR contains the length attribute of the column.
122. In DB2 UDB for z/OS and OS/390, and DB2 UDB for UWO, zoned decimal is not supported for local operations.
123. In DB2 UDB for UWO, DISPLAY SIGN LEADING SEPARATE is not supported.

CCSID Values in SQLDATA and 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 following must be true:

- the sixth byte of the SQLDAID in the SQLDA header is set to '+'
- the SQLNAME length is set to 8
- the first 4 bytes of SQLNAME are set as described in the Table 53.

In the DESCRIBE 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. If the column is a datetime column, the SQLDATA field of the SQLVAR can contain the CCSID of the string representation of the datetime value. The CCSID is located in bytes 3 and 4 as described in Table 53.

Table 53. CCSID values for SQLDATA[1] or SQLNAME

Data Type	Subtype	Bytes 1 & 2	Bytes 3 & 4
Character	SBCS data	X'0000'	ccsid
Character	Mixed data	X'0000'	ccsid
Character	Bit data	X'0000'	X'FFFF'[2]
Graphic	Not Applicable	X'0000'	ccsid
Datetime	Not Applicable	X'0000'	ccsid
Any other data type	Not Applicable	Not Applicable	Not Applicable

Notes:

G
G
G
1. In DB2 UDB for UWO, the value for SQLDATA does not follow this format on a 64-bit systems or systems using little endian integer formats. In these cases, the CCSID value can be returned by casting the value as an integer.

G
2. In DB2 UDB for UWO, X'0000' is returned instead of X'FFFF' for bit data.

INCLUDE SQLDA Declarations

For C

In C, INCLUDE SQLDA declarations are equivalent (but not necessarily identical) 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[SQLVAR2_PAD];
                }  len;
                char *sqldatalen;
                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 (Part 1 of 3)

```
/**********************************************************************/
/* Macros for using the sqlvar2 fields.                               */
/**********************************************************************/

/**********************************************************************/
/*   '2' in the 7th byte of sqldaid indicates a doubled 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 (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 (Part 3 of 3)

For COBOL

In COBOL, INCLUDE SQLDA declarations are equivalent[124] (but not necessarily identical) 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 12. INCLUDE SQLDA Declarations for COBOL

124. The line starting with SQLVAR OCCURS has a different value in the include for each platform with 409 repsenting the lowest value. If this value is too low, a portable application should code the SQLDA definition directly, specifying the value required by the application.

Appendix E. SQLSTATE Values—Common Return Codes

This appendix contains a summary of return codes called SQLSTATE values that are defined for the DB2 UDB SQL relational database products. SQLSTATE values are produced when an SQL statement is executed. The SQLSTATE values provide application programs with common return codes for common error conditions. Return codes from other database operations (such as commands) are not included.

This summary includes SQLSTATE values that cover existing conditions from all of the IBM DB2 UDB relational database products. Many of these conditions are product-specific. These values have been included for the convenience of application developers concerned with a distributed database environment where any of these values could be returned.

G
G

The SQLSTATE values are consistent with the SQLSTATE specifications contained in SQL 1999 Core standard.

Using SQLSTATE Values

An SQLSTATE value is a return code that indicates the outcome of the most recently executed SQL statement. The mechanism used to access SQLSTATE values depends on where the SQL statement is executed:

- In embedded applications other than Java, SQLSTATE values are returned in the last five bytes of the SQLCA or in a stand-alone SQLSTATE variable. For more information see, "SQL Return Codes" on page 264.
- In Java, SQLSTATE values are returned by using getSQLState() method. For more information see, Appendix J, "Coding SQL Statements in Java Applications" on page 627.

SQLSTATE values are designed so that application programs can test for specific conditions or classes of conditions.

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. Programmers who want to use SQLSTATE as the basis of their applications' return codes can define their own SQLSTATE classes or subclasses:

- 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.

The class code of an SQLSTATE value indicates whether the SQL statement was executed successfully (class codes 00 and 01) or unsuccessfully (all other class codes).

Table 1 identifies the SQLSTATE class codes used by DB2 UDB SQL and the SQL 1999 Core standard.

Table 1. SQLSTATE Class Codes

© Copyright IBM Corp. 1982, 2002

SQLSTATE Values

Class Code	Meaning	Subclass Code Table
00	Unqualified Successful Completion	Table 2
01	Warning	Table 3
02	No Data	Table 4
03	SQL Statement Not Yet Complete	Table 5
07	Dynamic SQL Error	Table 6
08	Connection Exception	Table 7
09	Triggered Action Exception	Table 8
0A	Feature Not Supported	Table 9
0D	Invalid Target Type Specification	Table 10
0E	Invalid Schema Name List Specification	Table 11
0F	Invalid Token	Table 12
0K	Resignal When Handler Not Active	Table 13
0W	Prohibited Statement Encountered During Trigger	Table 14
20	Case Not Found for Case Statement	Table 15
21	Cardinality Violation	Table 16
22	Data Exception	Table 17
23	Constraint Violation	Table 18
24	Invalid Cursor State	Table 19
25	Invalid Transaction State	Table 20
26	Invalid SQL Statement Identifier	Table 21
27	Triggered Data Change Violation	Table 22
28	Invalid Authorization Specification	Table 23
2D	Invalid Transaction Termination	Table 24
2E	Invalid Connection Name	Table 25
2F	SQL Function Exception	Table 26
34	Invalid Cursor Name	Table 27
36	Cursor Sensitivity Exception	Table 28
38	External Function Exception	Table 29
39	External Function Call Exception	Table 30
3B	Savepoint Exception	Table 31
3C	Ambiguous Cursor Name	Table 32
40	Transaction Rollback	Table 33
42	Syntax Error or Access Rule Violation	Table 34
44	WITH CHECK OPTION Violation	Table 35
46	Java Errors	Table 36
51	Invalid Application State	Table 37
53	Invalid Operand or Inconsistent Specification	Table 38
54	SQL or Product Limit Exceeded	Table 39
55	Object Not in Prerequisite State	Table 40
56	Miscellaneous SQL or Product Error	Table 41
57	Resource Not Available or Operator Intervention	Table 42

Class Code	Meaning	Subclass Code Table
58	System Error	Table 43

Table 2. Class Code 00: Unqualified Successful Completion

SQLSTATE	Meaning
00000	Execution of the SQL statement was successful and did not result in any type of warning or exception condition.

Table 3. Class Code 01: Warning

SQLSTATE Value	Meaning
01002	A DISCONNECT error occurred.
01003	Null values were eliminated from the argument of a column function.
01004	The value of a string was truncated when assigned to a host variable.
01005	Insufficient number of entries in an SQLDA.
01006	A privilege was not revoked.
01007	A privilege was not granted.
0100A	The query expression of the view is too long for the information schema.
0100C	One or more ad hoc result sets were returned from the procedure.
0100D	The cursor that was closed has been re-opened on the next result set within the chain.
0100E	The procedure returned too many result sets.
01503	The number of result columns is larger than the number of host variables provided.
01504	The UPDATE or DELETE statement does not include a WHERE clause.
01505	The statement was not executed because it is unacceptable in this environment.
01506	An adjustment was made to a DATE or TIMESTAMP value to correct an invalid date resulting from an arithmetic operation.
01507	One or more non-zero digits were eliminated from the fractional part of a number used as the operand of a multiply or divide operation.
01508	The statement was disqualified for blocking for reasons other than storage.
01509	Blocking was cancelled for a cursor because there is insufficient storage in the user virtual machine.
01510	Blocking was cancelled for a cursor because a blocking factor of at least two rows could not be maintained.
01511	Performance may not be optimum because of the number of predicates specified in the WHERE clause.
01512	The REVOKE operation has no effect on CONNECT privileges.
01513	A subsequent commit operation will revoke all EXECUTE privileges on the package except for that of the owner.
01514	The tablespace has been placed in the check-pending state.
01515	The null value has been assigned to a host variable, because the non-null value of the column is not within the range of the host variable.
01516	An inapplicable WITH GRANT OPTION has been ignored.
01517	A character that could not be converted was replaced with a substitute character.
01518	The definition of the table has been changed to incomplete.

01519	The null value has been assigned to a host variable, because a numeric value is out of range.
01520	The null value has been assigned to a host variable, because the characters cannot be converted.
01521	A specified server-name is undefined but is not needed until the statement is executed or the alias is used.
01522	The local table or view name used in the CREATE ALIAS statement is undefined.
01523	ALL was interpreted to exclude ALTER, INDEX, REFERENCES, and TRIGGER, because these privileges cannot be granted to a remote user.
01524	The result of a column function does not include the null values that were caused by evaluating the arithmetic expression implied by the column of the view.
01525	The number of INSERT values is not the same as the number of columns.
01526	Isolation level has been escalated.
01527	A SET statement references a special register that does not exist at the AS.
01528	WHERE NOT NULL is ignored, because the index key cannot contain null values.
01529	As a result of the DROP INDEX, the UNIQUE constraint is no longer enforced.
01530	Definition change may require a corresponding change on the read-only systems.
01532	An undefined object name was detected.
01533	An undefined column name was detected.
01534	The string representation of a datetime value is invalid.
01535	An arithmetic operation on a date or timestamp has a result that is not within the valid range of dates.
01536	During remote bind where existence checking is deferred, the server-name specified does not match the current server.
01537	An SQL statement cannot be EXPLAINed, because it references a remote object.
01538	The table cannot be subsequently defined as a dependent, because it has the maximum number of columns.
01539	Connection is successful but only SBCS characters should be used.
01540	A limit key has been truncated to 40 bytes.
01541	Operator command processing has completed successfully.
01542	Authorization ID does not have the privilege to perform the operation as specified.
01543	A duplicate constraint has been ignored.
01544	The null value has been assigned to a host variable, because a substring error occurred; for example, an argument of SUBSTR is out of range.
01545	An unqualified column name has been interpreted as a correlated reference.
01546	A column of the explanation table is improperly defined.
01547	A mixed data value is improperly formed.
01548	The authorization ID does not have the privilege to perform the specified operation on the identified object.
01550	The index was not created, because an index with the specified description already exists.
01551	A table in a partitioned tablespace is not available, because its partitioned index has not been created.
01552	An ambiguous qualified column name was resolved to the first of the duplicate names in the FROM clause.
01553	Isolation level RR conflicts with a tablespace locksize of page.
01554	Decimal multiplication may cause overflow.
01555	Mixed data is invalid and has been truncated according to SBCS rules.

01557	Too many host variables have been specified on SELECT INTO or FETCH.
01558	A distribution protocol has been violated.
01560	A redundant GRANT has been ignored.
01561	An update to a data capture table was not signaled to the originating subsystem.
01562	The new path to the log (newlogpath) in the database configuration file is invalid.
01563	The current path to the log file (logpath) is invalid. The log file path is reset to the default.
01564	The null value has been assigned to a host variable, because division by zero occurred.
01565	The null value has been assigned to a host variable, because a miscellaneous data exception occurred; for example, the character value for the CAST, DECIMAL, FLOAT, or INTEGER scalar function is invalid; a floating-point NAN (not a number) or invalid data in a packed decimal field was detected.
01566	The index has been placed in a pending state.
01567	The table was created but not journaled.
01568	The dynamic SQL statement ends with a semicolon.
01569	Statement has been successfully executed, but there may be some character conversion inconsistencies.
01570	The bind process detected a character string in an INSERT or UPDATE statement that is too large for the target column.
01571	The bind process detected a numeric value that is out of range.
01572	The bind process detected an invalid datetime format, such as an invalid string representation or an invalid value.
01573	The bind process detected a null insert or update value that is null for a column that cannot contain null values.
01574	The bind process detected an INSERT, UPDATE, or DELETE that is not permitted on this object.
01575	The bind process detected a non-updatable column in an INSERT or UPDATE statement.
01576	The bind process detected a CREATE INDEX statement for a view.
01577	The bind process detected a CREATE VIEW statement that includes an operator or operand that is not valid for views.
01578	The bind process detected operands of an operator that are not compatible.
01579	The bind process detected a numeric constant that is either too long or has a value that is not within the range of its data type.
01580	The bind process detected an update or insert value that is not compatible with the column.
01581	The bind process detected incompatible operands of a UNION operator.
01582	The bind process detected a string that is too long.
01583	The bind process detected a decimal divide operation that is invalid, because the result would have a negative scale.
01584	The bind process detected an insert or update value of a long string column that is neither a host variable nor NULL.
01585	The bind process detected a table that cannot be accessed, because it is inactive.
01586	Processing the statement resulted in one or more tables being automatically placed into a check pending state.
01587	The unit of work was committed or rolled back, but the outcome is not fully known at all sites.
01588	The LIKE predicate has an invalid escape character.
01589	A statement contains redundant specifications.
01590	Type 2 indexes do not have subpages.

01591	The result of the positioned UPDATE or DELETE may depend on the order of the rows.
01593	An ALTER TABLE may cause data truncation.
01594	Insufficient number of entries in an SQLDA for ALL information (i.e. not enough descriptors to return the distinct name).
01595	The view has replaced an existing, invalidated view.
01596	Comparison functions were not created for a distinct type based on a long string data type.
01597	Specific and non-specific volume IDs are not allowed in a storage group.
01598	An attempt has been made to activate an active event monitor or deactivate an inactive event monitor.
01599	Bind options were ignored on REBIND.
01600	SUBPAGES ignored on alter of catalog index.
01602	The optimization level has been reduced.
01603	CHECK DATA processing found constraint violations and moved them to exception tables.
01604	The SQL statement was explained and not executed.
01605	A recursive common table expression may contain an infinite loop.
01606	The node or system database directory is empty.
01607	The difference between the times on nodes in a read-only transactions exceed the defined threshold.
01608	An unsupported value has been replaced.
01611	The cursor that was closed has been re-opened on the next result set within the chain.
01612	The part clause of a LOCK TABLE statement is not valid.
01614	There are fewer locators than the number of result sets.
01616	The estimated CPU cost exceeds the resource limit.
01618	The ALTER NODEGROUP operation is not complete for all or some of the specified nodes.
01620	Some base tables of UNION ALL may be the same table.
01621	The retrieved LOB value may have been changed.
01622	Statement completed successfully but a system error occurred after the statement completed."
01623	The value of DEGREE is ignored.
01624	The GBPCACHE specification is ignored because the bufferpool does not allow caching.
01625	The schema name appears more than once in the CURRENT PATH.
01626	The database has only one bufferpool.
01627	The DATALINK value may not be valid because the table is in reconcile pending or reconcile is not a possible state.
01628	The user-specified access path hints are invalid. The access path hints are ignored.
01629	User-specified access path hints were used during access path selection.
01630	Virtual storage or database resource is not available.
01631	The external program could not be updated with the associated procedure or function attributes.
01632	The number of concurrent connections has exceeded the defined entitlement for the product.
01633	The materialized query table may not be used to optimize the processing of queries.
01634	The distinct data type name is too long and cannot be returned in the SQLDA. The short name is returned instead. of queries.
01635	Statements in the same program have duplicate QUERYNOs.
01636	Integrity of non-incremental data remains unverified by the database manager.

01637	Debugging is not enabled.
01638	SUBPAGES greater than one are not supported for Type 1 indexes in a data sharing environment.
01639	The federated object may require the invoker to have necessary privileges on data source objects.
01640	ROLLBACK TO SAVEPOINT occurred when there were uncommitted INSERTs or DELETEs that cannot be rolled back.
01642	Column not long enough for the largest possible USER default value.
01643	Assignment to SQLCODE or SQLSTATE variable does not signal a warning or error.
01644	DEFINE NO is not applicable for a lob space or data sets using the VCAT option.
01645	The executable for the SQL procedure is not saved in the catalog.
01646	A result sets could not be returned because the cursor was closed.
01647	A DB2SQL BEFORE trigger changed to DB2ROW.
01648	COMPRESS column attribute ignored because VALUE COMPRESSION has not been activated for the table.
01649	The bufferpool configuration has been completed but will not take effect until the next database restart.
01650	Index and table statistics are inconsistent.
01651	The event monitor was activated successfully, however some monitoring information may be lost.
01652	The isolation clause was ignored because of the statement context.
01653	The authorizations were granted to the user, but groups were not considered since the authorization name is more than 8 bytes.
01654	The buffer pool is not started.
01655	The event monitor was created successfully but at least one event monitor target table already exists.
01656	ROLLBACK TO savepoint caused a NOT LOGGED table space to be placed in the LPL.
01657	The buffer pool configuration will not take effect until the next database restart, due to insufficient memory.
01Hxx	Valid warning SQLSTATEs returned by a user-defined function or external procedure CALL.
01H51	An MQSeries Application Messaging Interface message was truncated.

Table 4. Class Code 02: No Data

SQLSTATE	Meaning
02000	One of the following exceptions occurred: • The result of the SELECT INTO statement or the subselect of the INSERT statement was an empty table. • The number of rows identified in the searched UPDATE or DELETE statement was zero. • The position of the cursor referenced in the FETCH statement was after the last row of the result table. • The fetch orientation is invalid.
02001	No additional result sets returned.
02502	Delete or update hole detected.

Table 5. Class Code 03: SQL Statement Not Yet Complete

SQLSTATE	Meaning
03000	Asynchronous execution is not yet completed.

Table 6. Class Code 07: Dynamic SQL Error

SQLSTATE	Meaning
07001	The number of host variables is not the same as the number of parameter markers.
07002	The call parameter list or control block is invalid.
07003	The statement identified in the EXECUTE statement is a select-statement, or is not in a prepared state.
07004	The USING clause is required for dynamic parameters.
07005	The statement name of the cursor identifies a prepared statement that cannot be associated with a cursor.
07006	An input host variable, transition variable, or parameter marker cannot be used, because of its data type.
07007	The dynamic statement requires a result area and none was specified.

Table 7. Class Code 08: Connection Exception

SQLSTATE	Meaning
08001	The application requester is unable to establish the connection.
08002	The connection already exists.
08003	The connection does not exist.
08004	The application server rejected establishment of the connection.
08007	Transaction resolution unknown.
08501	A DISCONNECT is not allowed when the connection uses an LU6.2 protected conversation.
08502	The CONNECT statement issued by an application process running with a SYNCPOINT of TWOPHASE has failed, because no transaction manager is available.
08504	An error was encountered while processing the path rename configuration file.

Table 8. Class Code 09: Triggered Action Exception

SQLSTATE	Meaning
09000	A triggered SQL statement failed.

Table 9. Class Code 0A: Feature Not Supported

SQLSTATE	Meaning
0A001	The CONNECT statement is invalid, because the process is not in the connectable state.
0A502	The action or operation is not enabled for this database instance.
0A503	Federated insert, update, or delete operation cannot be compiled because of potential data inconsistency.

Table 10. Class Code 0D: Invalid Target Type Specification

SQLSTATE	Meaning
0D000	The target structured data type specification is a proper subtype of the source structured data type.

Table 11. Class Code 0E: Invalid Schema Name List Specification

SQLSTATE	Meaning
0E000	The schema name list in a SET PATH statement is not valid.

Table 12. Class Code 0F: Invalid Token

SQLSTATE	Meaning
0F001	The locator value does not currently represent any value.

Table 13. Class Code 0K: Resignal When Handler Not Active

SQLSTATE	Meaning
0K000	A RESIGNAL was issued but a handler is not active.

Table 14. Class Code 0W: Prohibited Statement Encountered During Trigger

SQLSTATE	Meaning
0W000	The statement is not allowed in a trigger.

Table 15. Class Code 20: Case Not Found for Case Statement

SQLSTATE	Meaning
20000	The case was not found for the CASE statement.

Table 16. Class Code 21: Cardinality Violation

SQLSTATE	Meaning
21000	The result of a SELECT INTO, scalar fullselect, or subquery of a basic predicate is more than one value.
21501	A multiple-row INSERT into a self-referencing table is invalid.
21502	A multiple-row UPDATE of a primary key is invalid.
21504	A multiple-row DELETE from a self-referencing table with a delete rule of RESTRICT or SET NULL is invalid.
21505	A row function must return not more than one row.

Table 17. Class Code 22: Data Exception

SQLSTATE	Meaning
22001	Character data, right truncation occurred; for example, an update or insert value is a string that is too long for the column, or a datetime value cannot be assigned to a host variable, because it is too small.
22002	A null value, or the absence of an indicator parameter was detected; for example, the null value cannot be assigned to a host variable, because no indicator variable is specified.
22003	A numeric value is out of range.
22004	A null value cannot returned from a procedure that is defined as PARAMETER STYLE GENERAL or a type-preserving method that is invoked with a non-null argument.

SQLSTATE	Meaning
22006	The fetch orientation is invalid.
22007	An invalid datetime format was detected; that is, an invalid string representation or value was specified.
22008	Datetime field overflow occurred; for example, an arithmetic operation on a date or timestamp has a result that is not within the valid range of dates.
2200G	The most specific type does not match.
22011	A substring error occurred; for example, an argument of SUBSTR is out of range.
22012	Division by zero is invalid.
22018	The character value for the CAST, DECIMAL, FLOAT, or INTEGER scalar function is invalid.
22019	The LIKE predicate has an invalid escape character.
22021	A character is not in the coded character set.
22023	A parameter or host variable value is invalid.
22024	A NUL-terminated input host variable or parameter did not contain a NUL.
22025	The LIKE predicate string pattern contains an invalid occurrence of an escape character.
2202D	A null instance was used with a mutator method.
22501	The length control field of a variable length string is negative or greater than the maximum.
22503	The string representation of a name is invalid.
22504	A mixed data value is invalid.
22505	The local date or time length has been increased, but the executing program relies on the old length.
22506	A reference to a datetime special register is invalid, because the clock is malfunctioning or the operating system timezone parameter is out of range.
22508	CURRENT PACKAGESET is blank.
22511	ADT length exceeds maximum column length. The value for a ROWID or reference column is not valid.
22512	A host variable in a predicate is invalid, because its indicator variable is negative.
22519	The primary or foreign key cannot be activated.
22521	The foreign key cannot be defined, because the primary key of the parent table is inactive.
22522	A CCSID value is not valid at all, not valid for the data type or subtype, or not valid for the encoding scheme.
22524	Character conversion resulted in truncation
22525	Partitioning key value is not valid.
22526	A key transform function generated no rows or duplicate rows.
22527	Invalid input data detected for a multiple row insert.

Table 18. Class Code 23: Constraint Violation

SQLSTATE	Meaning
23001	The update or delete of a parent key is prevented by a RESTRICT update or delete rule.
23502	An insert or update value is null, but the column cannot contain null values.
23503	The insert or update value of a foreign key is invalid.
23504	The update or delete of a parent key is prevented by a NO ACTION update or delete rule.
23505	A violation of the constraint imposed by a unique index or a unique constraint occurred.

SQLSTATE	Meaning
23506	A violation of a constraint imposed by an edit or validation procedure occurred.
23507	A violation of a constraint imposed by a field procedure occurred.
23508	A violation of a constraint imposed by the DDL Registration Facility occurred.
23509	The owner of the package has constrained its use to environments which do not include that of the application process.
23510	A violation of a constraint on the use of the command imposed by the RLST table occurred.
23511	A parent row cannot be deleted, because the check constraint restricts the deletion.
23512	The check constraint cannot be added, because the table contains rows that do not satisfy the constraint definition.
23513	The resulting row of the INSERT or UPDATE does not conform to the check constraint definition.
23514	Check data processing has found constraint violations.
23515	The unique index could not be created or unique constraint added, because the table contains duplicate values of the specified key.
23520	The foreign key cannot be defined, because all of its values are not equal to a parent key of the parent table.
23521	The update of a catalog table violates an internal constraint.
23522	The range of values for the identity column or sequence is exhausted.

Table 19. Class Code 24: Invalid Cursor State

SQLSTATE	Meaning
24501	The identified cursor is not open.
24502	The cursor identified in an OPEN statement is already open.
24503	The cursor identified in the PUT statement is a select cursor, or the cursor identified in the FETCH statement is an insert cursor.
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row.
24505	COMMIT is invalid, because blocking is in effect and an insert cursor is open.
24506	The statement identified in the PREPARE is the statement of an open cursor.
24507	FETCH CURRENT was specified, but the current row is deleted, or a value of an ORDER BY column of the current row has changed.
24510	An UPDATE or DELETE operation was attempted against a delete or update hole
24513	FETCH NEXT, PRIOR, CURRENT, or RELATIVE is not allowed, because the cursor position is not known.
24514	A previous error has disabled this cursor.
24516	A cursor has already been assigned to a result set.
24517	A cursor was left open by an external function or method.
24518	A cursor is not defined to handle row sets, but a rowset was requested.
24519	A hole was detected on a multiple row FETCH statement, but indicator variables were not provided.
24520	The cursor identified in the UPDATE or DELETE statement is not positioned on a rowset.
24521	A positioned DELETE or UPDATE statement specified a row of a rowset, but the row is not contained within the current rowset.
24522	The fetch orientation is inconsistent with the definition of the cursor and whether rowsets are supported for the cursor.

SQLSTATE	Meaning
24523	The width of a rowset cursor has not yet been set.

Table 20. Class Code 25: Invalid Transaction State

SQLSTATE	Meaning
25000	An update operation is invalid for the application execution environment.
25006	An update operation is not valid because the transaction is read only.
25501	The statement is only allowed as the first statement in a unit of work.

Table 21. Class Code 26: Invalid SQL Statement Identifier

SQLSTATE	Meaning
26501	The statement identified does not exist.
26505	An extended EXECUTE, DECLARE CURSOR, or DESCRIBE has been issued against an empty section.
26507	An extended EXECUTE with an OUTPUT DESCRIPTOR has been issued against a section that is not a Single Row SELECT.
26508	The statement identified in an extended PREPARE Single Row is not a select-statement.
26510	The statement name specified in a DECLARE CURSOR already has a cursor allocated to it.

Table 22. Class Code 27: Triggered Data Change Violation

SQLSTATE	Meaning
27000	An attempt was made to change the same row in the same table more than once in the same SQL statement.

Table 23. Class Code 28: Invalid Authorization Specification

SQLSTATE	Meaning
28000	Authorization name is invalid.

Table 24. Class Code 2D: Invalid Transaction Termination

SQLSTATE	Meaning
2D521	SQL COMMIT or ROLLBACK are invalid in the current operating environment.
2D522	COMMIT and ROLLBACK are not allowed in an ATOMIC Compound statement.
2D528	Dynamic COMMIT or COMMIT ON RETURN procedure is invalid for the application execution environment
2D529	Dynamic ROLLBACK is invalid for the application execution environment.

Table 25. Class Code 2E: Invalid Connection Name

SQLSTATE	Meaning
2E000	Connection name is invalid.

Table 26. Class Code 2F: SQL Function Exception

SQLSTATE	Meaning
2F002	The SQL function attempted to modify data, but the function was not defined as MODIFIES SQL DATA.
2F003	The statement is not allowed in a function or procedure.
2F004	The SQL function attempted to read data, but the function was not defined as READS SQL DATA.
2F005	The function did not execute a RETURN statement.

Table 27. Class Code 34: Invalid Cursor Name

SQLSTATE	Meaning
34000	Cursor name is invalid.

Table 28. Class Code 36: Cursor Sensitivity Exception

SQLSTATE	Meaning
36001	A SENSITIVE cursor cannot be defined for the specified select-statement.

Table 29. Class Code 38: External Function Exception

SQLSTATE	Meaning
38xxx	Valid error SQLSTATEs returned by an external routine or trigger.
38001	The external routine is not allowed to execute SQL statements.
38002	The external routine attempted to modify data, but the routine was not defined as MODIFIES SQL DATA.
38003	The statement is not allowed in a routine.
38004	The external routine attempted to read data, but the routine was not defined as READS SQL DATA.
38501	Error occurred while calling a user-defined function, external procedure, or trigger (using the SIMPLE CALL or SIMPLE CALL WITH NULLS calling convention).
38502	The external function is not allowed to execute SQL statements.
38503	A user-defined function or procedure has abnormally terminated (abend).
38504	A user-defined function has been interrupted by the user to stop a probable looping condition.
38505	An SQL statement is not allowed in a routine on a FINAL CALL.
38506	Function failed with error from OLE DB provider.
38552	A function in the SYSFUN schema has terminated with an error.
38553	A function in a system schema has terminated with an error.
38H01	An MQSeries function failed to initialize.
38H02	MQSeries Application Messaging Interface failed to terminate the session.
38H03	MQSeries Application Messaging Interface failed to properly process a message.
38H04	MQSeries Application Messaging Interface failed in sending a message.
38H05	MQSeries Application Messaging Interface failed to read/receive a message.
38H06	An MQSeries Application Messaging Interface message was truncated.

SQLSTATE Values

Table 30. Class Code 39: External Function Call Exception

SQLSTATE	Meaning
39001	A user-defined function has returned an invalid SQLSTATE.
39004	A null value is not allowed for an IN or INOUT argument when using PARAMETER STYLE GENERAL.
39501	An output argument value returned from a function or a procedure was too long.
39502	An output SQLDA from a procedure was incorrectly modified.

Table 31. Class Code 3B: Savepoint Exception

SQLSTATE	Meaning
3B001	The savepoint is not valid.
3B002	The maximum number of savepoints has been reached.
3B501	A duplicate savepoint name was detected.
3B502	A RELEASE or ROLLBACK TO SAVEPOINT was specified, but a savepoint does not exist.
3B503	A SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT is not allowed in a trigger, function, or global transaction.
3B504	A SAVEPOINT is not allowed because a resource is registered that does not support savepoints.

Table 32. Class Code 3C: Ambiguous Cursor Name

SQLSTATE	Meaning
3C000	The cursor name is ambiguous.

Table 33. Class Code 40: Transaction Rollback

SQLSTATE	Meaning
40001	Deadlock or timeout with automatic rollback occurred.
40003	The statement completion is unknown.
40503	A private dbspace is in use by another application process.
40504	A system error has caused the unit of work to be rolled back.
40506	The current transaction was rolled back because of an SQL error.
40507	The current transaction was rolled backed as a result of a failure creating an index.

Table 34. Class Code 42: Syntax Error or Access Rule Violation

SQLSTATE	Meaning
42501	The authorization ID does not have the privilege to perform the specified operation on the identified object.
42502	The authorization ID does not have the privilege to perform the operation as specified.
42503	The authorization ID specified in SET CURRENT SQLID is not one of the authorization IDs of the application process.
42504	A specified privilege cannot be revoked from a specified authorization-name.
42505	Connection authorization failure occurred.
42506	Owner authorization failure occurred.

SQLSTATE	Meaning
42508	The specified database privileges cannot be granted to PUBLIC.
42509	SQL statement is not authorized, because of the DYNAMICRULES option.
42510	The authorization ID does not have the privilege to create functions or procedures in the WLM environment.
42511	The authorization ID does not have the privilege to retrieve the DATALINK value.
42601	A character, token, or clause is invalid or missing.
42602	A character that is invalid in a name has been detected.
42603	An unterminated string constant has been detected.
42604	An invalid numeric or string constant has been detected.
42605	The number of arguments specified for a scalar function is invalid.
42606	An invalid hexadecimal constant has been detected.
42607	An operand of a column function or CONCAT operator is invalid.
42608	The use of NULL or DEFAULT in VALUES is invalid.
42609	All operands of an operator or predicate are parameter markers.
42610	A parameter marker is not allowed.
42611	The column or argument definition is invalid.
42612	The statement string is an SQL statement that is not acceptable in the context in which it is presented.
42613	Clauses are mutually exclusive.
42614	A duplicate keyword is invalid.
42615	An invalid alternative was detected.
42616	Invalid options are specified.
42617	The statement string is blank or empty.
42618	A host variable is not allowed.
42620	Read-only SCROLL was specified with the UPDATE clause.
42621	The check constraint or generated column expression is invalid.
42622	A name or label is too long.
42623	A DEFAULT clause cannot be specified.
42625	A CASE expression is invalid.
42626	A column specification is not allowed for a CREATE INDEX that is built on an auxiliary table.
42627	RETURNS clause must be specified prior to predicate specification using the EXPRESSION AS clause.
42628	A TO SQL or FROM SQL transform function is defined more than once in a transform definition.
42629	Parameter names must be specified for SQL routines.
42630	An SQLSTATE or SQLCODE variable is not valid in this context.
42631	An expression must be specified on a RETURN statement in an SQL function.
42632	There must be a RETURN statement in an SQL function or method.
42701	A duplicate column name in an INSERT or UPDATE operation or the SET transition-variable was detected.
42702	A column reference is ambiguous, because of duplicate names.
42703	An undefined column or parameter name was detected.
42704	An undefined object or constraint name was detected.

SQLSTATE Values

SQLSTATE	Meaning
42705	An undefined server-name was detected.
42706	Column names in ORDER BY are invalid, because all columns of the result table are unnamed.
42707	A column name in ORDER BY does not identify a column of the result table.
42708	The locale specified in a SET LOCALE or locale sensitive function was not found.
42709	A duplicate column name in a PRIMARY, UNIQUE, or FOREIGN KEY clause was detected.
42710	A duplicate object or constraint name was detected.
42711	A duplicate column name was detected in the object definition or ALTER TABLE statement.
42712	A duplicate table designator was detected in the FROM clause or REFERENCING clause of a CREATE TRIGGER statement.
42713	A duplicate object was detected in a list of objects.
42714	A host variable can be defined only once.
42716	Any function called within the body of an inline function must already be defined.
42718	The local server name is not defined.
42720	The nodename for the remote database was not found in the node directory.
42721	The special register name is unknown at the server.
42723	A function with the same signature already exists in the schema.
42724	Unable to access an external program used for a user-defined function or a procedure.
42725	A routine or method was referenced directly (not by either signature or by specific instance name), but there is more than one specific instance of that routine or method.
42726	Duplicate names for common table expressions were detected.
42727	No default primary tablespace exists for the new table.
42728	A duplicate node was detected in the nodegroup definition.
42729	The node is not defined.
42730	The container name is already used by another tablespace.
42731	The container name is already used by this tablespace.
42732	A duplicate schema name in the SET CURRENT PATH statement was detected.
42733	A procedure with the specified name cannot be added to the schema because the procedure overloading is not allowed in this database and there is already a procedure with the same name in the schema.
42734	A duplicate parameter-name, SQL variable name, label, or condition-name was detected.
42735	The nodegroup for the table space is not defined for the buffer pool.
42736	The label specified on the GOTO, ITERATE, or LEAVE statement is not found or not valid.
42737	The condition specified is not defined.
42738	A duplicate column name or unnamed column was specified in a DECLARE CURSOR statement of a FOR statement.
42739	A duplicate transform was detected.
42740	No transforms were found for the specified type. No transforms were dropped.
42741	A transform group is not defined for a data type.
42742	A subtable or subview of the same type already exists in the typed table or typed view hierarchy.
42743	The search method is not found in the index extension.
42744	A TO SQL or FROM SQL transform function is not defined in a transform group.
42745	The routine would define a overriding relationship with an existing method.

SQLSTATE	Meaning
42746	A method name cannot be the same as a structured type name within the same type hierarchy.
42801	Isolation level UR is invalid, because the result table is not read-only.
42802	The number of insert or update values is not the same as the number of columns.
42803	A column reference in the SELECT or HAVING clause is invalid, because it is not a grouping column; or a column reference in the GROUP BY clause is invalid.
42804	The result expressions in a CASE expression are not compatible.
42805	An integer in the ORDER BY clause does not identify a column of the result table.
42806	A value cannot be assigned to a host variable, because the data types are not compatible.
42807	The INSERT, UPDATE, or DELETE is not permitted on this object.
42808	A column identified in the INSERT or UPDATE operation is not updatable.
42809	The identified object is not the type of object to which the statement applies.
42810	A view is identified in a FOREIGN KEY clause.
42811	The number of columns specified is not the same as the number of columns in the SELECT clause.
42812	A library name is required in CREATE TABLE in the system naming mode.
42813	WITH CHECK OPTION cannot be used for the specified view.
42814	The column cannot be dropped, because it is the only column in the table.
42815	The data type, length, scale, value, or CCSID is invalid.
42816	A datetime value or duration in an expression is invalid.
42817	The column cannot be dropped, because RESTRICT was specified and a view or constraint is dependent on the column.
42818	The operands of an operator or function are not compatible.
42819	An operand of an arithmetic operation or an operand of a function that requires a number is not a number.
42820	A numeric constant is too long, or it has a value that is not within the range of its data type.
42821	A data type for an assignment to a column or variable is not compatible with the data type.
42822	An expression in the ORDER BY clause or GROUP BY clause is not valid.
42823	Multiple columns are returned from a subquery that only allows one column.
42824	An operand of LIKE is not a string, or the first operand is not a column.
42825	The rows of UNION, INTERSECT, EXCEPT, or VALUES do not have compatible columns.
42826	The rows of UNION, INTERSECT, EXCEPT, or VALUES do not have the same number of columns.
42827	The table identified in the UPDATE or DELETE is not the same table designated by the cursor.
42828	The table designated by the cursor of the UPDATE or DELETE statement cannot be modified, or the cursor is read-only.
42829	FOR UPDATE OF is invalid, because the result table designated by the cursor cannot be modified.
42830	The foreign key does not conform to the description of the parent key.
42831	A column of a primary key, unique key, or ROWID does not allow null values.
42832	The operation is not allowed on system objects.
42833	The qualified object name is inconsistent with the naming option.
42834	SET NULL cannot be specified, because the foreign key does not allow null values.
42835	Cyclic references cannot be specified between named derived tables.
42836	The specification of a recursive, named derived table is invalid.

SQLSTATE Values

SQLSTATE	Meaning
42837	The column cannot be altered, because its attributes are not compatible with the current column attributes.
42838	An invalid use of a tablespace was detected.
42839	Indexes and long columns cannot be in separate tablespaces from the table.
42841	A parameter marker can not be a user-defined type or reference type.
42842	A column or parameter definition is invalid, because a specified option is inconsistent with the column description.
42844	A function in a select list item has produced a BOOLEAN result.
42845	An invalid use of a NOT DETERMINISTIC or EXTERNAL ACTION function was detected.
42846	Cast from source type to target type is not supported.
42847	An OVRDBF command was issued for one of the referenced files, but one of the parameters is not valid for SQL.
42848	Isolation level CS WITH KEEP LOCKS is not allowed.
42849	The specified option is not supported for external routines.
42850	A logical file is invalid in CREATE VIEW.
42851	A referenced file is not a table, view, or physical file.
42852	The privileges specified in GRANT or REVOKE are invalid or inconsistent. (For example, GRANT ALTER on a view.)
42853	Both alternatives of an option were specified, or the same option was specified more than once.
42854	A result column data type in the select list is not compatible with the defined type in a typed view or materialized query table definition.
42855	The assignment of the LOB to this host variable is not allowed. The target host variable for all fetches of this LOB value for This cursor must be a locator or LOB variable.
42856	The alter of a CCSID to the specified CCSID is not valid.
42857	A referenced file has more than one format.
42858	Operation cannot be applied to the specified object.
42860	The constraint cannot be dropped because it is enforcing a primary key or ROWID.
42862	An extended dynamic statement cannot be executed against a non-extended dynamic package.
42863	An undefined host variable in REXX has been detected.
42866	The data type in either the RETURNS clause or the CAST FROM clause in the CREATE FUNCTION statement is not appropriate for the data type returned from the sourced function or RETURN statement in the function body.
42872	FETCH statement clauses are incompatible with the cursor definition.
42873	An invalid number of rows was specified in a multiple-row FETCH or multiple-row INSERT.
42874	ALWCPYDTA(*NO) was specified, but a copy is necessary to implement the select-statement.
42875	The schema-name portion of a qualified name must be the same name as the schema name.
42876	Different CCSIDs for keys in CREATE INDEX are only allowed with a *HEX sort sequence.
42877	The column name cannot be qualified.
42878	An invalid function or procedure name was used with the EXTERNAL keyword.
42879	The data type of one or more input parameters in the CREATE FUNCTION statement is not appropriate for the corresponding data type in the source function.
42880	The CAST TO and CAST FROM data types are incompatible, or would always result in truncation of a fixed string.
42881	Invalid use of a function.

SQLSTATE	Meaning
42882	The specific instance name qualifier is not equal to the function name qualifier.
42883	No function or method was found with a matching signature.
42884	No routine was found with the specified name and compatible arguments.
42885	The number of input parameters specified on a CREATE FUNCTION statement does not match the number provided by the function named in the SOURCE clause.
42886	The IN, OUT, or INOUT parameter attributes do not match.
42887	The function is not valid in the context where it occurs.
42888	The table does not have a primary key.
42889	The table already has a primary key.
42890	A column list was specified in the references clause, but the identified parent table does not have a unique constraint with the specified column names.
42891	A duplicate UNIQUE constraint already exists.
42892	The referential constraint and trigger are not allowed, because the DELETE rule and trigger event are not compatible.
42893	The object or constraint cannot be dropped or authorities cannot be revoked from the object, because other objects are dependent on it.
42894	The value of a column or sequence attribute is invalid.
42895	For static SQL, an input host variable cannot be used, because its data type is not compatible with the parameter of a procedure or user-defined function.
42896	The ASP number is invalid.
42898	An invalid correlated reference or transition table was detected in a trigger.
42899	Correlated references and column names are not allowed for triggered actions with the FOR EACH STATEMENT clause.
428A0	An error occurred with the sourced function on which the user-defined function is based.
428A1	Unable to access a file referenced by a file reference variable.
428A2	A table cannot be assigned to a multi-node node group, because it does not have a partition key.
428A3	An invalid path has been specified for an event monitor.
428A4	An invalid value has been specified for an event monitor option.
428A5	An exception table named in a SET INTEGRITY statement either does not have the proper structure, or it has been defined with generated columns, constraints or triggers.
428A6	An exception table named in a SET CONSTRAINTS statement cannot be the same as one of the tables being checked.
428A7	There is a mismatch in the number of tables being checked and in the number of exception tables specified in the SET CONSTRAINTS statement.
428A8	Cannot reset the check-pending state using the SET CONSTRAINTS statement on a descendent table while a parent table is in the check-pending state.
428A9	The node range is invalid.
428AA	The column name is not a valid column for an event monitor table.
428B0	Nesting not valid in ROLLUP, CUBE, or GROUPING SETs.
428B1	Incorrect number of table space container specifications that are not designated for specific nodes.
428B2	The path name for the container is not valid.
428B3	An invalid SQLSTATE was specified.
428B4	The part clause of a LOCK TABLE statement is not valid.
428B7	A number specified in an SQL statement is out of the valid range.

SQLSTATE Values

SQLSTATE	Meaning
428B8	The name specified on a rename is not valid.
428C0	The node cannot be dropped, because it is the only node in the nodegroup.
428C1	Only one ROWID or IDENTITY column can be specified for a table.
428C2	Examination of the function body indicates that the given clause should have been specified on the CREATE FUNCTION statement.
428C3	The language specified for a subtype must be the same as that of its supertype.
428C4	The number of elements on each side of the predicate operator is not the same.
428C5	No data type mapping was found for a data type from the data source.
428C6	The item references in a SET statement must all be transition variables or none of the item references must be transition variables.
428C7	A ROWID or reference column specification is not valid.
428C8	The expression cannot be cast to a ROWID or reference type.
428C9	A ROWID or IDENTITY column cannot be specified as the target column of an INSERT or UPDATE.
428CA	A table in append mode cannot have a clustered index.
428CB	The pagesize for a table space must match the page size of the associated bufferpool.
428D1	Unable to access a file referenced by a DATALINK value.
428D2	AS LOCATOR cannot be specified for a non-LOB parameter.
428D3	GENERATED was specified with a data type that is not a ROWID or a distinct type based on a ROWID.
428D4	A cursor specified in a FOR statement cannot be referenced in an OPEN, CLOSE, or FETCH statement.
428D5	The ending label does not match the beginning label.
428D6	UNDO is not allowed for NOT ATOMIC compound statements.
428D7	The condition value is not allowed.
428D8	The sqlcode or sqlstate variable declaration is not valid.
428D9	The table specified in the host variable in the LIKE clause is not compatible with the table specified in the LIKE clause.
428DB	An object is not valid as a supertype, supertable, or superview.
428DC	The function or method is not valid as the transform for this type.
428DE	The PAGESIZE value is not supported.
428DF	The data types specified in CREATE CAST are not valid.
428DG	The function specified in CREATE CAST is not valid.
428DH	The operation is not valid for typed tables.
428DJ	The options associated with an inherited column cannot be changed.
428DK	The scope for the reference column is already defined.
428DL	The parameter of an external or sourced function has a scope defined.
428DM	The scope table or view is not valid for the reference type.
428DN	SCOPE is not specified in the RETURNS clause of an external function or is specified in the RETURNS clause of a sourced function.
428DP	The type is not a structured type.
428DQ	A subtable or subview cannot have a different schema name than its supertable or superview.
428DR	Operation cannot be applied to a subtable.

SQLSTATE	Meaning
428DS	An index on the specified columns cannot be defined on the subtable.
428DT	The operand of an expression is not a valid scoped reference type.
428DU	A type is not included in the required type hierarchy.
428DV	The left operand of a dereference operator is not valid.
428DW	Object identifier column cannot be referenced using the dereference operator.
428DX	Object identifier column is required to define the root table or root view of a typed table or typed view hierarchy.
428DY	Table statistics cannot be updated for a subtable.
428DZ	An object identifier column cannot be updated.
428E0	The definition of the index does not match the definition of the index extension.
428E1	The result of the range-producing table function is inconsistent with that of the key transformation table function for the index extension.
428E2	The number or the type of key-target parameters does not match the number or type of key transform function for the index extension.
428E3	The argument for the function in an index extension is not valid.
428E4	The function is not supported in an CREATE INDEX EXTENSION statement.
428E5	SELECTIVITY clause can only be specified with a user-defined predicate.
428E6	The argument of the search method in the user-defined predicate does not match the one in the corresponding search method of the index extension.
428E7	The type of the operand following the comparison operator in the user-defined predicate does not match the RETURNS data type.
428E8	A search target or search argument parameter does not match a parameter name of the function being created.
428E9	An argument parameter name cannot appear as both a search target and search argument in the same exploitation rule.
428EA	A fullselect in a typed view is not valid.
428EB	A column in a subview cannot be read only when the corresponding column in the superview is updatable.
428EC	The fullselect specified for the materialized query table is not valid.
428ED	Structured types with Datalink or Reference type attributes cannot be constructed.
428EE	Option not valid for remote data source.
428EF	Value for the option is not valid remote data source.
428EG	Missing required option for remote data source.
428EH	Option is already defined for remote data source.
428EJ	Option is not defined so cannot be set for remote data source.
428EK	The qualifier for a declared global temporary table name or an index on a declared global temporary table must be SESSION.
428EL	A transform function not valid for use with a function or method.
428EM	The TRANSFORM GROUP clause is required.
428EN	A transform group is specified that is not used.
428EP	A structured type cannot depend on itself either directly or indirectly.
428EQ	The returns type of the routine is not the same as the subject type.
428ER	A method specification cannot be dropped before the method body is dropped.

SQLSTATE Values

SQLSTATE	Meaning
428ES	A method body does not correspond to the language type of the method specification.
428ET	INLINE LENGTH value is not valid.
428EU	TYPE or VERSION is not specified in the server definition.
428EV	Pass-through facility is not supported for the type of data source.
428EW	The table cannot be converted to or from a materialized query table.
428EX	Routine cannot be used as a transform function because it is either a built-in function or a method.
428EY	The data type of the search target in a user-defined predicate does not match the data type of the source key of the specified index extension.
428EZ	A window specification for an OLAP function is not valid.
428F0	A ROW function must include at least two columns.
428F1	An SQL TABLE function must return a table result.
428F2	An integer expression must be specified on a RETURN statement in an SQL procedure.
428F4	The SENSITIVITY specified on FETCH is not allowed for the cursor.
428F5	The invocation of a function is ambiguous.
428F6	Cursor is scrollable, but the result table involves output from a table function.
428F7	The operation was attempted on an external routine, but the operation is only allowed on an SQL routine.
428F9	A sequence expression cannot be specified in this context.
428FA	The scale of the decimal number must be zero.
428FC	The length of the encryption password is not valid.
428FD	The password used for decryption does not match the password used to encrypt the data.
428FE	The data is not a result of the ENCRYPT function.
428FF	The buffer pool specification is not valid.
428FG	The table used to define a staging table is not valid.
428FH	The materialized query table option is not valid.
428FI	The ORDER OF clause was specified, but the referenced table designator is not ordered.
428FJ	ORDER BY is not allowed in the outer fullselect of a view or materialized query table.
428FM	An INSERT statement within a SELECT specified a view which is not a symmetric view.
428FN	ALLOW FULL REFRESH must be specified for altering this view.
428FO	The ALTER VIEW failed because the fullselect is invalid.
428FP	Only one INSTEAD OF trigger is allowed for each kind of operation on a view.
428FQ	An INSTEAD OF trigger must not specify a view that is defined using WITH CHECK OPTION or a view that is defined on another view that is defined WITH CHECK OPTION.
428FR	A column cannot be altered as specified.
428FS	A column cannot be added to an index.
428FT	Partitioning clauses cannot be specified on a non-partitioned table or materialized query table.
428FU	Built-in type returned from the FROM SQL transform function or method does not match the corresponding built-in type for the TO SQL transform function or method.
428FV	The method cannot be defined as an overriding method.
42901	A column function does not include a column name.
42902	The object of the INSERT, UPDATE, or DELETE is also identified (possibly implicitly through a view) in a FROM clause.

SQLSTATE	Meaning
42903	A WHERE, VALUES, GROUP BY, HAVING, or SET clause includes an invalid reference, such as a column or OLAP function.
42904	The SQL procedure was not created because of a compile error.
42905	DISTINCT is specified more than once in a subselect.
42906	A column function in a subquery of a HAVING clause includes an expression that applies an operator to a correlated reference.
42907	The string is too long.
42908	The statement does not include a required column list.
42909	CREATE VIEW includes an operator or operand that is not valid for views. For example, UNION or UNION ALL.
42910	The statement is not allowed in a Compound statement.
42911	A decimal divide operation is invalid, because the result would have a negative scale.
42912	A column cannot be updated, because it is not identified in the UPDATE clause of the select-statement of the cursor.
42913	An UPDATE or DELETE WHERE CURRENT OF that is invalid has been detected.
42914	The DELETE is invalid, because a table referenced in a subquery can be affected by the operation.
42915	An invalid referential constraint has been detected.
42916	The alias cannot be created, because it would result in a repetitive chain of aliases.
42917	The object cannot be explicitly dropped or altered.
42918	A user-defined data type cannot be created with a system-defined data type name (for example, INTEGER).
42919	Nested compound statements are not allowed.
42920	A GROUP BY or HAVING clause is implicitly or explicitly specified in a SELECT INTO or a subquery of a basic predicate.
42921	Containers cannot be added to the tablespace.
42922	DROP SCHEMA cannot be executed under commitment control.
42923	Program or package must be recreated to reference an alias-name.
42924	An alias resolved to another alias rather than a table or view at the remote location.
42925	Recursive named derived tables cannot specify SELECT DISTINCT and must specify UNION ALL.
42926	Locators are not allowed with COMMIT(*NONE).
42927	The function cannot be altered to NOT DETERMINISTIC or EXTERNAL ACTION because it is referenced by one or more existing views.
42928	WITH EMPTY TABLE cannot be specified.
42930	The same column was identified in FOR UPDATE OF and ORDER BY.
42932	The program preparation assumptions are incorrect.
42937	The parameter must not have a subtype of mixed.
42939	The name cannot be used, because the specified identifier is reserved for system use.
42943	An empty non-modifiable package cannot be committed.
42944	The authorization ID cannot be both an owner and primary group owner.
42945	ALTER CCSID is not allowed on a tablespace or database that contains a view.
42961	The server name specified does not match the current server.
42962	A long column, LOB column, structured type column or datalink column cannot be used in an index, a key, or a constraint.

SQLSTATE Values

SQLSTATE	Meaning
42968	The connection failed, because there is no current software license.
42969	The package was not created and the current unit of work was rolled back, because of internal limitations or an invalid section number.
42970	COMMIT HOLD or ROLLBACK HOLD is not allowed to a non-application server.
42971	SQL statements cannot be executed under commitment control, because commitment control is already active to another relational database.
42972	An expression in a join-condition references columns in more than one of the operand tables.
42977	The authorization ID cannot be changed when connecting to the local server.
42978	An indicator variable is not a small integer.
42981	CREATE SCHEMA is not allowed if changes are pending in the unit of work.
42984	The privilege cannot be granted to the view, because *OBJOPR or *OBJMGT authority exists on a dependent view or table, and the grantee does not have *ALLOBJ or the specified privilege on the dependent table or view.
42985	The statement is not allowed in a routine.
42986	The source table in a RENAME TABLE statement is referenced in a view, materialized query table, trigger, or constraint.
42987	The statement is not allowed in a trigger.
42988	The operation is not allowed with mixed ASCII data.
42989	A GENERATED column that is based on an expression and cannot used in a BEFORE trigger.
42990	A unique index or unique constraint is not allowed because the key columns are not a superset of the partitioned key columns.
42991	The BOOLEAN data type is currently only supported internally.
42993	The column, as defined, is too large to be logged.
42994	Raw device containers are not currently supported on this system.
42995	The requested function does not apply to global temporary tables.
42996	The partition key cannot be a datetime or floating-point column.
42997	Capability is not supported by this version of DB2.
42998	A referential constraint is not allowed because the foreign key columns are not a superset of the partitioned key columns or the node group is not the same as the parent table.
429A0	A foreign key cannot reference a parent table defined as ″not logged initially″.
429A1	The nodegroup is not valid for the table space.
429A2	The row type within a row reference must be the same as the row type of the target table.
429A3	Row type cannot be directly used as the type of a column. Only references to row types are allowed.
429A5	A row type can not be added to a table that already has a row type.
429A6	The table identified in a row reference operation does not have a row type.
429A7	Function with the READS SQL DATA property cannot be used in the specified context.
429A8	Distinct types cannot be based on either REF types or ADTs.
429A9	SQL statement cannot be processed by DataJoiner.
429AA	The ″not logged initially″ attribute cannot be activated.
429B1	A stored procedure specifying COMMIT ON RETURN cannot be the target of a nested CALL statement.
429B2	The specified inline length value for the structured type or column is too small.

SQLSTATE	Meaning
429B3	The object may not be defined on a subtable.
429B4	The data filter function cannot be a LANGUAGE SQL function.
429B5	The data type of the instance parameter in the index extension is not valid in the same exploitation rule.
429B6	Rows from a distributed table cannot be redistributed because the table contains a datalink column with FILE LINK CONTROL.
429B7	A referential constraint with a delete rule of CASCADE is not allowed on a table with a DataLink column with FILE LINK CONTROL.
429B8	A routine defined with PARAMETER STYLE JAVA cannot have a structured type as a parameter or returns type.
429B9	DEFAULT or NULL cannot be used in an attribute assignment.
429BA	The FEDERATED keyword must be used with a reference to a federated database object.
429BB	Data type of parameter or SQL variable is not supported in SQL routine.
429BC	There are multiple conflicting container operations in the ALTER TABLESPACE statement.
429BD	RETURN must be the last SQL statement of the atomic compound statement within an SQL row or table function.
429BE	The primary key or a unique key is a subset of the columns in the dimensions clause.

Table 35. Class Code 44: WITH CHECK OPTION Violation

SQLSTATE	Meaning
44000	The INSERT or UPDATE is not allowed, because a resulting row does not satisfy the view definition.

Table 36. Class Code 46: Java Errors

SQLSTATE	Meaning
46001	The URL specified on an install or replace of a jar procedure did not identify a valid jar file.
46002	The jar name specified on the install, replace, or remove of a Java procedure is not valid.
46003	The jar file cannot be removed, a class is in use by a procedure.
46007	A Java function has a Java method with an invalid signature.
46008	A Java function could not map to a single Java method.
46103	A Java routine encountered a ClassNotFound exception.
46501	The install or remove jar procedure for ″<jar-id>″ specified the use of a deployment descriptor.
46502	A user-defined procedure has returned a DYNAMIC RESULT SET of an invalid class. The parameter is not a DB2 result set.

Table 37. Class Code 51: Invalid Application State

SQLSTATE	Meaning
51002	The package corresponding to an SQL statement execution request was not found.
51003	Consistency tokens do not match.
51004	An address in the SQLDA is invalid.
51005	The previous system error has disabled this function.

SQLSTATE	Meaning
51006	A valid connection has not been established.
51008	The release number of the precompiled program is not valid.
51009	COMMIT or ROLLBACK is not allowed, because commitment control has not been started.
51010	The programmable interface for operator commands is not valid when within a unit of work.
51012	The index has been marked invalid.
51013	An attempt has been made to use an index that has been marked invalid.
51015	An attempt was made to execute a section that was found to be in error at bind time.
51016	A package or view cannot be rebound, because the character set under which it was originally prepared is different than the character set under which the database manager is running.
51017	The user is not logged on.
51021	SQL statements cannot be executed until the application process executes a rollback operation.
51022	A CONNECT that specifies an authorization name is invalid when a connection (either current or dormant) already exists to the server named in that CONNECT statement.
51023	The database is already in use by another instance of the database manager.
51024	A view cannot be used, because it has been marked inoperative.
51025	An application in the XA transaction processing environment is not bound with SYNCPOINT TWOPHASE.
51026	An event monitor cannot be turned on, because its target path is already in use by another event monitor.
51027	The IMMEDIATE CHECKED option of the SET CONSTRAINTS statement is not valid since a table is not in the check-pending state.
51028	A package cannot be used, because it is marked inoperative.
51030	The procedure referenced in a DESCRIBE PROCEDURE, ASSOCIATE LOCATOR, or an ALLOCATE CURSOR statement has not yet been called within the application process.
51032	A valid CCSID has not yet been specified for this DB2 UDB for OS/390 and z/OS subsystem.
51033	The operation is not allowed because it operates on a result set that was not created by the current server.
51034	The routine defined with MODIFIES SQL DATA is not valid in the context in which it is invoked.
51036	An implicit connect to a remote server is not allowed because a savepoint is outstanding.
51037	The operation is not allowed because a trigger has been marked inoperative.
51038	SQL Statements may no longer be issued by the routine.
51039	The ENCRYPTION PASSWORD value is not set.

Table 38. Class Code 53: Invalid Operand or Inconsistent Specification

SQLSTATE	Meaning
53001	A clause is invalid, because the tablespace is a workfile.
53004	DSNDB07 is the implicit workfile database.
53014	The specified OBID is invalid.
53022	Host variable or parameter is not allowed.
53035	Key limits must be specified in the CREATE or ALTER INDEX statement.
53036	The number of PART specifications is not the same as the number of partitions.
53037	A partitioned index cannot be created on a table in a non-partitioned tablespace.

SQLSTATE	Meaning
53038	The number of key limit values is zero or greater than the number of columns in the key.
53039	The PART clause of the ALTER statement is omitted or invalid.
53040	The bufferpool cannot be changed as specified.
53041	Only 4K buffer pools can be used for an index.
53043	Columns with different field procedures cannot be compared.
53044	The columns have a field procedure, but the field types are not compatible.
53045	The data type of the key limit constant is not the same as the data type of the column.
53060	Public dbspaces must be acquired from a recoverable storage pool.
53088	LOCKMAX is inconsistent with the specified LOCKSIZE.
53089	The number of host variable parameters for a stored procedure is not equal to the number of expected host variable parameters.
53090	Data encoded with different encoding schemes cannot be referenced in the same SQL statement.
53091	The encoding scheme specified is not the same as the encoding scheme currently in use for the containing tablespace.
53092	Type 1 index cannot be created for a table using the ASCII encoding scheme.
53093	The CCSID ASCII or UNICODE clause is not supported for this database or tablespace.
53094	The PLAN_TABLE cannot be created with the FOR ASCII clause.
53095	CREATE or ALTER statement cannot define an object with the specified encoding scheme.
53096	The PART clause was specified on CREATE AUXILIARY TABLE, but the base table is not partitioned.
53097	LOBs cannot be specified as parameters when the NO WLM ENVIRONMENT is specified.
53098	The auxiliary table cannot be created because a column was specified that is not a LOB column.
53099	A WLM ENVIRONMENT name must be specified on the CREATE FUNCTION statement.
530A0	An ALTER TABLE statement specified a precision and scale that is not as large as the existing precision and scale.
530A1	An ALTER TABLE statement specified FLOAT as the new data type for a column, but there is an existing index or constraint that restricts the use of FLOAT.
530A2	The VALUES clause is not allowed on the specified index.

Table 39. Class Code 54: SQL or Product Limit Exceeded

SQLSTATE	Meaning
54001	The statement is too long or too complex.
54002	A string constant is too long.
54004	The statement has too many table names or too many items in a SELECT or INSERT list.
54005	The sort key is too long, or has too many columns.
54006	The result of concatenation is too long.
54008	The key is too long, a column of the key is too long, or the key many columns.
54009	Too many users were specified in GRANT or REVOKE.
54010	The record length of the table is too long.
54011	Too many columns were specified for a table, view, or table function.
54012	The FIELDPROC literal list is too long.
54013	The statement has too many host variables.

SQLSTATE	Meaning
54014	Too many cursors are open in a unit of work.
54015	A section was not created as a result of executing the null form of an extended dynamic PREPARE, or preprocessing a PREPARE statement.
54016	No more tables can be created in this dbspace.
54017	The maximum number of active packages for a unit of work has been exceeded.
54018	The row is too long.
54019	The maximum number of late descriptors has been exceeded, probably because too many different CCSIDs were used.
54020	No more indexes can be created for this table.
54021	Too many constraints, or the size of the constraint is too large.
54023	The limit for the number of parameters or arguments for a function or a procedure has been exceeded.
54024	The check constraint is too long.
54025	The table description exceeds the maximum size of the object descriptor.
54027	The catalog has the maximum number of user-defined indexes.
54028	The maximum number of concurrent LOB handles has been reached.
54029	The maximum number of open directory scans has been reached.
54030	The maximum number of event monitors are already active.
54031	The maximum number of files have already been assigned the event monitor.
54032	The maximum size of a table has been reached.
54033	The maximum number of partitioning maps has been reached.
54034	The combined length of all container names for the tablespace is too long.
54035	An internal object limit exceeded.
54036	The path name for the container is too long.
54037	The container map for the tablespace is too complicated.
54038	Maximum depth of nested routines or triggers was exceeded.
54039	The container size is too small or too large.
54040	Too many references to transition variables and transition tab columns or the row length for these references is too long.
54041	Only 32767 OBIDs are allowed.
54042	Only one index is allowed on an auxiliary table.
54044	A multiple-byte (UCS-2) sort sequence table cannot be supported in DRDA because it is too large.
54045	The maximum level of a type hierarchy has been reached.
54046	The maximum allowable parameters is exceeded in an index extension.
54047	The maximum size of a table space is exceeded.
54048	A temporary table space with sufficient page size does not exist.
54049	Length of an instance of a structured type exceeds the system limit.
54050	The maximum allowable attributes is exceeded in a structured type.
54051	Value specified on FETCH ABSOLUTE or RELATIVE is invalid.
54052	The number of block pages for a buffer pool is too large for the buffer pool.
54053	The value specified for BLOCKSIZE is not in the valid range.

SQLSTATE	Meaning
54054	The combination of the number of table space partitions and the corresponding length of the partitioning limit key it too large.
54055	The maximum number of versions has been reached for a table or index. The maximum number of versions has been reached for a table or index.

Table 40. Class Code 55: Object Not in Prerequisite State

SQLSTATE	Meaning
55001	The database must be migrated.
55002	The explanation table is not defined properly.
55003	The DDL registration table is not defined properly.
55004	The database cannot be accessed, because it is no longer a shared database.
55005	Recursion is not supported to a non- application server.
55006	The object cannot be dropped, because it is currently in use by the same application process.
55007	The object cannot be altered, because it is currently in use by the same application process.
55009	The system attempted to write to a read-only file or a write-protected diskette.
55011	The operation is disallowed, because the workfile database is not in the stopped state.
55012	A clustering index already exists on the table.
55014	The table does not have an index to enforce the uniqueness of the primary key.
55015	The ALTER statement cannot be executed, because the pageset is not in the stopped state.
55016	The ALTER statement is invalid, because the pageset has user-managed data sets.
55017	The table cannot be created in the tablespace, because it already contains a table.
55018	The schema cannot be dropped, because it is in the library list.
55019	The table is in an invalid state for the operation.
55020	A work file database is already defined for the member.
55021	Change of data type or length of host variable is invalid, because blocking is in effect.
55022	The file server is not registered with this database.
55023	An error occurred calling a procedure.
55024	The tablespace cannot be dropped, because data related to a table is also in another tablespace.
55025	The database must be restarted.
55026	A temporary tablespace cannot be dropped.
55027	The current unit of work is only prepared to process a COMMIT or ROLLBACK statement.
55028	Parameter in the LASTING GLOBALV file is either missing or incorrect.
55029	Local program attempted to connect to a remote database.
55030	A package specified in a remote BIND REPLACE operation must not have a system list.
55031	The format of the error mapping file is incorrect.
55032	The CONNECT statement is invalid, because the database manager was stopped after this application was started.
55033	An event monitor cannot be activated in the same unit of work in which it is created or modified.
55034	The event monitor is in an invalid state for the operation.
55035	The table cannot be dropped, because it is protected.
55036	The node cannot be dropped, because it has not been removed from the partitioning map.

SQLSTATE	Meaning
55037	The partitioning key cannot be dropped, because the table is in a multi-node nodegroup.
55038	The nodegroup cannot be used, because it is being rebalanced.
55039	The access or state transition is not allowed, because the tablespace is not in an appropriate state.
55040	The database's split image is in the suspended state.
55041	Containers cannot be added to a tablespace while a rebalance is in progress.
55042	The alias is not allowed because it identifies a single member of a multiple member file.
55043	The attributes of a structured type cannot be altered when a typed table of typed view based on the type exists.
55044	The PROCEDURE must have a status of STOP-REJ, or the PSERVER must be stopped with IMPL=N, before it can be altered or dropped.
55045	The SQL Archive (SAR) file for the routine cannot be created because a necessary component is not available at the server.
55046	The specified SQL archive (SAR) does not match the target environment.
55047	A routine declared as NOT FEDERATED attempted to access a federated object.
55048	Encrypted data cannot be encrypted.
55049	The event monitor table is not properly defined.
55050	An object cannot be created into a protected schema.
55051	The ALTER BUFFERPOOL statement is currently in progress.
55052	A TABLE or TABLESPACE cannot be altered to NOT LOGGED when there are uncommitted changes that have been made to the table or tablespace.
55053	A CREATE or ALTER statement specified a NOT LOGGED INITIALLY clause, but the table space contains other tables.
55054	A method cannot be called recursively.

Table 41. Class Code 56: Miscellaneous SQL or Product Error

SQLSTATE	Meaning
56004	The statement failed, because the Invalid Entities table is full.
56010	The subtype of a string variable is not the same as the subtype at bind time, and the difference cannot be resolved by character conversion.
56016	The partitioning keys are not specified in ascending or descending order.
56018	A column cannot be added to the table, because it has an edit procedure.
56023	An invalid reference to a remote object has been detected.
56025	An invalid use of AT ALL LOCATIONS in GRANT or REVOKE has been detected.
56027	A nullable column of a foreign key with a delete rule of SET NULL cannot be part of the key of a partitioned index.
56031	The clause or scalar function is invalid, because mixed and DBCS data are not supported on this system.
56033	The insert or update value of a long string column must be a host variable or NULL.
56034	ALLUSERS can only be used in GRANT CONNECT without a password.
56035	Referential constraints cannot cross dbspaces resident in different types of storage pools.
56036	Specific and non-specific volume IDs are not allowed in a storage group.
56038	The requested feature is not supported in this environment.
56040	CURRENT SQLID cannot be used in a statement that references remote objects.

SQLSTATE	Meaning
56041	An Extended PREPARE can only be executed using the DRDA protocol if it has an input SQLDA.
56042	Only one package can be created or modified in a unit of work, and, while that package is being created or modified, all statements in that unit of work must be issued against that package. If the package is non-modifiable, only Extended PREPARE statements can be issued.
56044	An attempt was made to execute a section that has been marked invalid in a modifiable package that is undergoing modification.
56045	The application must issue a rollback operation to back out the change that was made at the read-only application server.
56046	CREATE PACKAGE with the REPLACE option cannot be issued against a modifiable package.
56047	PREPARE Adding Empty Section was not preceded by a CREATE PACKAGE with the NOMODIFY option.
56048	Three-part package names are not supported.
56049	An unexpected error occurred when attempting to rebind a view with a new version of the database manager. The view must be dropped and recreated.
56052	The remote requester tried to bind, rebind, or free a trigger package.
56053	The parent of a table in a read-only shared database must also be a table in a read-only shared database.
56054	User-defined datasets for objects in a shared database must be defined with SHAREOPTIONS(1,3).
56055	The database is defined as SHARE READ, but the tablespace or indexspace has not been defined on the owning system.
56056	The description of an object in a SHARE READ database must be consistent with its description in the OWNER system.
56057	A database cannot be altered from SHARE READ to SHARE OWNER.
56058	A COMMIT WORK statement or a ROLLBACK WORK statement cannot be dynamically prepared or executed.
56059	An error occurred when binding a triggered SQL statement.
56060	An LE function failed.
56062	A distributed operation is invalid, because the unit of work was started before DDF.
56063	In Single User Mode only one CICS task can issue an SQL statement.
56064	The bind operation is disallowed, because the program depends on functions of a release from which fallback has occurred.
56065	The bind operation is disallowed, because the DBRM has been modified or was created for a different release.
56066	The rebind operation is disallowed, because the plan or package depends on functions of a release from which fallback has occurred.
56067	The rebind operation is disallowed, because the value of SYSPACKAGE.IBMREQD is invalid.
56076	A DB2 Server for VSE & VM application requestor that uses DRDA-only protocols cannot be connected to a DB2 Server for VSE & VM application server that uses SQLDS-only protocols.
56079	Neither protocol option AUTO nor DRDA can be specified, because the DRDA facility has not been installed for the application requester.
56080	The data type is not allowed in DB2 private protocol processing.
56082	The statement cannot be executed, because it identifies a DB2 system that does not support character conversion.
56084	An unsupported SQLTYPE was encountered in a select list or input list.
56088	ALTER FUNCTION failed because functions cannot modify data when they are processed in parallel.

SQLSTATE Values

SQLSTATE	Meaning
56089	Specified option requires type 2 indexes.
56090	The type of the index cannot be changed.
56091	Multiple errors occurred as a result of executing a compound SQL statement.
56092	The type of authorization cannot be determined, because the authorization name is both a user id and group id.
56093	A query includes a column with a data type not supported by the application requestor.
56095	A bind option is invalid.
56096	Bind options are incompatible.
56097	LONG VARCHAR and LONG VARGRAPHIC column are not permitted in tablespaces using DEVICES.
56098	An error occurred during an implicit rebind or recompile.
56099	The REAL data type is not supported by the target database.
560A0	Action on a LOB value failed.
560A1	The tablespace name is not valid.
560A2	A LOB table and its associated base table space must be in the same database.
560A3	The table is not compatible with the database.
560A4	The operation is not allowed on an auxiliary table.
560A5	An auxiliary table already exists for the specified column or partition.
560A6	A table cannot have a LOB column unless it also has a ROWID column.
560A7	GBPCACHE NONE cannot be specified for a tablespace or index in GRECP.
560A8	An 8K or 16K bufferpool pagesize is invalid for a WORKFILE object.
560A9	An unsupported option was specified.
560AA	The clause or scalar function is invalid, because UCS-2 is not supported on this system.
560AB	The data type is not supported in an SQL routine.
560AC	Wrapper definition cannot be used for the specified type or version of data source.
560AD	A view name was specified after LIKE in addition to the INCLUDING IDENTITY COLUMN ATTRIBUTES clause.
560AE	A view was specified for LIKE, but it includes a ROWID column.
560AF	Prepare statement is not supported when using gateway concentrator.
560B0	Invalid new size value for table space container resizing.
560B1	Procedure failed because a result set was scrollable but the cursor was not positioned before the first row.
560B2	Open failed because the cursor is scrollable but the client does not support scrollable cursors.
560B3	Procedure failed because one or more result sets returned by the procedure are scrollable but the client does not support scrollable cursors.
560B4	Procedure failed because one or more result sets returned by the procedure are scrollable but hop sites are involved.
560B5	Local special register is not valid as used.
560B6	CALL is not allowed in an embedded ATOMIC compound statement.
560B7	For a multiple row INSERT, the usage of a sequence expression must be the same for each row.
560B8	The SQL statement cannot be executed because it was precompiled at a level that is incompatible with the current value of the ENCODING bind option or special register.
560B9	Hexadecimal constant GX is not allowed.

SQLSTATE	Meaning
560BA	ACTIVATE NOT LOGGED INITIALLY not supported for the specified table.
560BB	The same host variable must be used in both the USING and INTO clauses for an INOUT parameter in a dynamically prepared CALL statement.
560BC	An error has occurred when accessing the configuration file.
560BD	Unexpected error code received from data source.

Table 42. Class Code 57: Resource Not Available or Operator Intervention

SQLSTATE	Meaning
57001	The table is unavailable, because it does not have a primary index.
57002	GRANT and REVOKE are invalid, because authorization has been disabled.
57003	The specified bufferpool has not been activated.
57004	The table is unavailable, because it lacks a partitioned index.
57005	The statement cannot be executed, because a utility or a governor time limit was exceeded.
57006	The object cannot be created, because a DROP or CREATE is pending.
57007	The object cannot be used, because a DROP or ALTER is pending.
57008	The date or time local format exit has not been installed.
57009	Virtual storage or database resource is temporarily unavailable.
57010	A field procedure could not be loaded.
57011	Virtual storage or database resource is not available.
57012	A non-database resource is not available. This will not affect the successful execution of subsequent statements.
57013	A non-database resource is not available. This will affect the successful execution of subsequent statements.
57014	Processing was canceled as requested.
57015	Connection to the local DB2 not established.
57016	The table cannot be accessed, because it is inactive.
57017	Character conversion is not defined.
57018	A DDL registration table or its unique index does not exist.
57019	The statement was not successful, because of a problem with a resource.
57020	The drive containing the database is locked.
57021	The diskette drive door is open.
57022	The table could not be created, because the authorization ID of the statement does not own any suitable dbspaces.
57023	The DDL statement cannot be executed, because a DROP is pending of a DDL registration table.
57024	No appropriate CMS message repository can be accessed.
57025	There is not enough room in the dbspace(s) allocated to hold packages.
57026	The system dbspace SYS002 does not exist. This dbspace is used to store packages.
57027	The connection to the application server has been severed by the operator.
57028	The unit of work has been rolled back due to an excessive number of system wide lock requests.
57029	The unit of work has been rolled back due to an excessive number of lock requests by the unit of work.
57030	Connection to application server would exceed the installation-defined limit.

SQLSTATE	Meaning
57031	Connection to the application server is not possible, because the DB2 Server for VSE & VM virtual machine does not have access to that application server.
57032	The maximum number of concurrent databases have already been started.
57033	Deadlock or timeout occurred without automatic rollback.
57036	The transaction log does not belong to the current database.
57037	The ACQUIRE DBSPACE statement failed, because all storage pools for available dbspaces are full.
57038	No space is available in the storage pool.
57039	The VSE Online Resource Manager has been shut down, either by the operator, or due to a serious error.
57040	The communications directory was either not found, or it has the wrong file type.
57042	DDM recursion has occurred.
57043	A local SQL application program cannot be executed on an application server.
57044	The resource adapter cannot find an entry for the character set in the ASISSCR MACRO file.
57045	The resource adapter cannot find an entry for the character set in the SYSCHARSETS file.
57046	A new transaction cannot start because the database or instance is quiesced.
57047	An internal database file cannot be created, because the directory is not accessible.
57048	An error occurred while accessing a container for a tablespace.
57049	The operating system process limit has been reached.
57050	The file server is not currently available.
57051	The estimated CPU cost exceeds the resource limit.
57052	The node is unavailable, because it does not have containers for all temporary table spaces.
57053	A table is not available in a routine or trigger because of violated nested SQL statement rules.
57054	A table is not available until the auxiliary tables and indexes for its externally stored columns have been created.
57055	A temporary table space with sufficient page size was not available.
57056	The package is not available because the database is in NO PACKAGE LOCK mode.
57057	The SQL statement cannot be executed due to a prior condition in a DRDA chain of SQL statements.
57059	There is not enough space in the table space for the specified action.

Table 43. Class Code 58: System Error

SQLSTATE	Meaning
58001	The database cannot be created, because the assigned DBID is a duplicate.
58002	An exit has returned an error or invalid data.
58003	An invalid section number was detected.
58004	A system error (that does not necessarily preclude the successful execution of subsequent SQL statements) occurred.
58005	A system error (that prevents the successful execution of subsequent SQL statements) occurred.
58006	A system error occurred during connection.
58007	A system error occurred with datalink file management.
58008	Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements.

SQLSTATE	Meaning
58009	Execution failed due to a distribution protocol error that caused deallocation of the conversation.
58010	Execution failed due to a distribution protocol error that will affect the successful execution of subsequent DDM commands or SQL statements.
58011	The DDM command is invalid while the bind process in progress.
58012	The bind process with the specified package name and consistency token is not active.
58013	The SQLCODE is inconsistent with the reply message.
58014	The DDM command is not supported.
58015	The DDM object is not supported.
58016	The DDM parameter is not supported.
58017	The DDM parameter value is not supported.
58018	The DDM reply message is not supported.
58021	A system error occurred while loading a program.
58023	A system error has caused the current program to be canceled.
58024	An error has occurred in the underlying operating system.
58025	A column in a catalog table has the wrong data type.
58026	The number of host variables in the statement is not equal to the number of host variables in SQLSTTVRB.
58027	The package was not created and unit of work was rolled back due to an earlier system error.
58028	The commit operation failed, because a resource in the unit of work was not able to commit its resources.
58029	An internal error has occurred while attempting to log user data.
58030	An I/O error has occurred.
58031	The connection was unsuccessful, because of a system error.
58032	Unable to use the process for a fenced mode user-defined function.
58033	An unexpected error occurred while attempting to access a client driver.
58034	An error was detected while attempting to find pages for an object in a DMS tablespace.
58035	An error was detected while attempting to free pages for an object in a DMS tablespace.
58036	The internal tablespace ID specified does not exist.

Appendix F. 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 23.

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.

© Copyright IBM Corp. 1982, 2002

CCSID Values

Table 54. 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 for UWO, 13488 is only used to tag the GRAPHIC column of eucJP and eucTW databases.

Table 55. 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 56. 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 57. 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
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

Table 57. 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 58. 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

Table 58. 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
1125	Ukraine	X	C	X	C	C	C	C	C	C	C
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								
9056	Arabic (Storage Interchange)	X	C								
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

Table 58. 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
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

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

CCSID Values

Table 59. 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

Table 60. 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								
9066	Thailand (extended)	X	C								

CCSID Values

Table 61. 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

Table 62. 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

Table 63. 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

Table 64. 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		C								
5039	Japan HP-UX (J15)	X		C	C	X	C	C	C	C	C
5050	Japan (EUC)		C								
5052	Japan 2022 TCP		C								
5053	Japan 2022 TCP		C								

Table 64. 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
5054	Japan 2022 TCP		C								
5055	Japan 2022 TCP		C								
17354	Korea 2022 TCP		C								
25546	Korea 2022 TCP		C								
33722	Japan EUC		C								

Appendix G. 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 "CONNECT (Type 1)" on page 296
- CONNECT (Type 2) is used for distributed unit of work. See "CONNECT (Type 2)" on page 300

The following table summarizes the differences between CONNECT (Type 1) and CONNECT (Type 2) rules:

Table 65. CONNECT (Type 1) and CONNECT (Type 2) Differences

Type 1 Rules	Type 2 Rules
CONNECT statements can only be executed when the application process is in the connectable state. No more than one CONNECT statement can be executed within the same unit of work.	More than one CONNECT statement can be executed within the same unit of work. There are no rules about the connectable state.
G If the CONNECT statement fails because the server name is not listed in the local directory, the connection state of the application process is product-specific. If a CONNECT statement fails because the application process is not in the connectable state, the connection status of the application process is unchanged. If a CONNECT statement fails for any other reason, the application process is placed in the unconnected state.	If a CONNECT statement fails, the current connection is unchanged and any subsequent SQL statements are executed by the current server.
CONNECT ends its only existing connection of the application process. Accordingly, CONNECT also closes any open cursors of the application process. (The only cursors that can possibly be open when CONNECT is successfully executed are those defined with the WITH HOLD option.)	CONNECT does not end connections and does not close cursors.
A CONNECT to the current server is executed like any other CONNECT (Type 1) statement.	A CONNECT to the current server causes an error.[125]

Determining the CONNECT Rules That Apply

G A program preparation option is used to specify the type of CONNECT that will be performed by a program. The program preparation option is product-specific.

125. In DB2 UDB for z/OS and OS/390, the SQLRULES(STD) bind option can be used to allow a CONNECT to the current server.

© Copyright IBM Corp. 1982, 2002

The CONNECT rules that apply to an application process are determined by the first CONNECT statement that is executed (successfully or unsuccessfully) by that application process:

- If it is a CONNECT (Type 1), then CONNECT (Type 1) rules apply and CONNECT (Type 2) statements are invalid
- If it is a CONNECT (Type 2), then CONNECT (Type 2) rules apply and CONNECT (Type 1) statements are invalid.

Programs containing CONNECT statements that are precompiled with different CONNECT program preparation options cannot execute as part of the same application process. An error will occur when an attempt is made to execute the invalid CONNECT statement.

Connecting to Application Servers That Only Support Remote Unit of Work

CONNECT (Type 2) connections to application servers that only support remote unit of work might result in connections that are read-only.

If a CONNECT (Type 2) is performed to an application server that only supports 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 an application server that supports distributed unit of work:

- The connection allows read-only operations when there are dormant connections that allow updates to application servers that only support remote unit of work. In this case, the connection allows updates as soon as the dormant connecton is ended.
- Otherwise, the connection allows updates.

Appendix H. Coding SQL Statements in C Applications

This section describes the programming techniques that are unique to coding SQL statements within a C program. Throughout this book, C is used to represent either C or C++, except where explicitly noted otherwise.

Defining the SQL Communications Area in C

A C program that contains SQL statements must include one or both of the following:

- An SQLSTATE variable[126] declared as `char SQLSTATE[6]`
- An SQLCODE variable[126] declared as `long SQLCODE`

or,

- An SQLCA (which contains an SQLSTATE and SQLCODE variable).

The SQLSTATE and SQLCODE values are set by the database manager after each SQL statement is executed. An application can check the SQLSTATE or SQLCODE value to determine whether the last SQL statement was successful. See Appendix E, "SQLSTATE Values—Common Return Codes" on page 539 for more information.

The SQLCA can be coded in a C program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLCA;
```

A standard declaration includes both a structure definition and a static data area named 'sqlca'. The SQLCA must not be defined within an SQL declare section. See Appendix C, "SQL Communication Area (SQLCA)" on page 525 and "INCLUDE" on page 421 for more information.

The SQLSTATE, SQLCODE, and SQLCA variables must appear before any executable statements. The scope of the declaration must include the scope of all SQL statements in the program.

Note: Many SQL error messages contain message data that is of varying length. The lengths of these data fields are embedded in the value of the SQLCA sqlerrmc field. Because of these lengths, printing the value of sqlerrmc from a C program might give unpredictable results.

Defining SQL Descriptor Areas in C

The following statements require an SQLDA:

EXECUTE...USING DESCRIPTOR *descriptor-name*

FETCH...USING DESCRIPTOR *descriptor-name*

OPEN...USING DESCRIPTOR *descriptor-name*

DESCRIBE *statement-name* INTO *descriptor-name*

126. In DB2 UDB for z/OS and OS/390, the STDSQL(YES) option must be in effect to declare the SQLSTATE and SQLCODE variables. In DB2 UDB for UWO, the LANGLEVEL SQL92E option must be used to declare the SQLSTATE and SQLCODE variables.

© Copyright IBM Corp. 1982, 2002

PREPARE *statement-name* INTO *descriptor-name*

CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. An SQLDA can be coded in a C program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLDA;
```

A standard declaration includes only a structure definition with the name 'sqlda'. The SQLDA must not be defined within an SQL declare section. See Appendix D, "SQL Descriptor Area (SQLDA)" on page 529 and "INCLUDE" on page 421 for more information.

One benefit from using the INCLUDE SQLDA SQL statement is the following macro definition:

```
#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1)* sizeof(struc sqlvar))
```

This macro makes it easy to allocate storage for an SQLDA with a specified number of SQLVAR elements. In the following example, the SQLDASIZE macro is used to allocate storage for an SQLDA with 20 SQLVAR elements.

```
#include <stdlib.h>
EXEC SQL INCLUDE SQLDA;
struct sqlda *mydaptr;
short numvars = 20;
  .
  .
mydaptr = (struct sqlda *) malloc(SQLDASIZE(numvars));
mydaptr->sqln = 20;
```

Here are other macro definitions that are included with the INCLUDE SQLDA statement:

GETSQLDOUBLED(daptr)
: Returns 1 if the SQLDA pointed to by daptr has been doubled, or 0 if it has not been doubled. The SQLDA is doubled if the seventh byte in the SQLDAID field is set to '2'.

SETSQLDOUBLED(daptr, newvalue)
: Sets the seventh byte of SQLDAID to newvalue.

GETSQLDALONGLEN(daptr,n)
: Returns the length attribute of the *n*th entry in the SQLDA to which daptr points. Use this only if the SQLDA was doubled and the *n*th SQLVAR entry has a LOB data type.

SETSQLDALONGLEN(daptr,n,len)
: Sets the SQLLONGLEN field of the SQLDA to which daptr points to len for the *n*th entry. Use this only if the SQLDA was doubled and the *n*th SQLVAR entry has a LOB data type.

GETSQLDALENPTR(daptr,n)
: Returns a pointer to the actual length of the data for the *n*th entry in the SQLDA to which daptr points. The SQLDATALEN pointer field returns a pointer to a long (4 byte) integer. If the SQLDATALEN pointer is zero, a NULL pointer is returned. Use this only if the SQLDA has been doubled.

SETSQLDALENPTR(daptr,n,ptr)
Sets a pointer to the actual length of the data for the *n*th entry in the SQLDA to which daptr points. Use this only if the SQLDA has been doubled.

SQLDA declarations can appear wherever a structure definition is allowed. Normal C scope rules apply.

Embedding SQL Statements in C

SQL statements can be coded in a C program wherever executable statements can appear.

Each SQL statement in a C program must begin with EXEC SQL and end with a semicolon (;). [127] The EXEC SQL keywords must appear all on one line, but the remainder of the statement can appear on the next and subsequent lines.

For example, an UPDATE statement coded in a C program might be coded as follows:

```
EXEC SQL
   UPDATE DEPARTMENT
   SET MGRNO = :MGR_NUM
   WHERE DEPTNO = :INT_DEPT ;
```

Comments

In addition to SQL comments (--), C comments (/* ... */) can be included within embedded SQL statements wherever a blank is allowed, except between the keywords EXEC and SQL. C Comments can span any number of lines but cannot be nested. [128] Single-line comments (starting with //) can be used in a C++ source program but are not permitted anywhere in a C source program.

Continuation for SQL Statements

SQL statements can be contained on one or more lines. An SQL statement can be split wherever a blank can appear. A character-string constant or delimited identifier can be continued on the following line using the backslash (\). Identifiers that are not delimited cannot be continued. For graphic-string constants in EBCDIC, see product documentation.

Including Code

SQL statements or C statements can be included by embedding the following SQL statement at the point in the source code where the statements are to be embedded:

```
EXEC SQL INCLUDE name;
```

C #include statements cannot be used to include SQL statements or declarations of C variables that are referenced in SQL statements.

127. In DB2 UDB for z/OS and OS/390, if the HOST(C(FOLD)) option is specified, SQL keywords and SQL identifiers are folded to uppercase. When the option is not specified, SQL keywords must be specified in uppercase. For either case, host variables are never folded.

128. In DB2 UDB for z/OS and OS/390, the STDSQL(YES) option must be in effect to use SQL comments.

Margins

SQL statements must be coded in columns 1 through 80. [129]

Names

Any valid C variable name can be used for a host variable, as long as it:

- does not contain DBCS characters
- is less than or equal to 128 characters in length
- does not begin with 'DB2', 'DSN', 'RDI', or 'SQL' in any combination of uppercase or lowercase letters (these names are reserved for the database manager).

Access plan names must not start with 'DSN'. External entry names must not start with 'DSN', 'RDI', or 'SQL'.

For information on the length of a host identifier, see Table 37 on page 509.

NULLs and NULs

C and SQL both use the word null, but for different meanings. The C language has a null character (NUL), a null pointer (NULL), and a null statement (just a semicolon). The C NUL is a single character which compares equal to 0. The C NULL is a special reserved pointer value that does not point to any valid data object. The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a (nonnull) value.

Statement Labels

Executable SQL statements can be preceded with a label, if desired.

Preprocessor Considerations

The precompiler does not support C preprocessor directives.

Trigraphs

Some characters from the C character set are not available on all keyboards. These characters can be entered into a C source program using a sequence of three characters called a *trigraph*. Trigraphs are not supported within SQL statement, however, the following trigraph sequences are supported within host variable declarations:

??(left bracket ([)

??) right bracket (])

??< left brace ({)

??> right brace (})

??/ backslash (\)

129. In DB2 UDB for z/OS and OS/390, a program preparation option must be used to specify margins 1 and 80. If the program preparation option is not specified, the margins will be 1 and 72.

Handling SQL Errors and Warnings in C

The SQL WHENEVER statement tests the result of every SQL statement within its scope for an error or warning condition. The target for the GOTO clause in a WHENEVER statement must be within the scope of any SQL statements affected by the WHENEVER statement.

The stand-alone SQLSTATE and SQLCODE or information in the SQLCA can also be used in the detection or further handling of error and warning conditions. See Appendix C, "SQL Communication Area (SQLCA)" on page 525 for more information.

Using Host Variables in C

All host variables used in SQL statements must be explicitly declared. A host variable used in an SQL statement must be declared prior to the first use of the host variable in an SQL statement.

The C statements that are used to define the host variables must be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement.

All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.

An SQL statement that uses a host variable must be within the scope of the statement in which the variable was declared.

Host variables must not be unions, union elements, or pointers. However, a single pointer can be used to reference an SQLDA. Host variables must not be arrays or array elements unless they are used to represent indicator arrays or indicator variables.

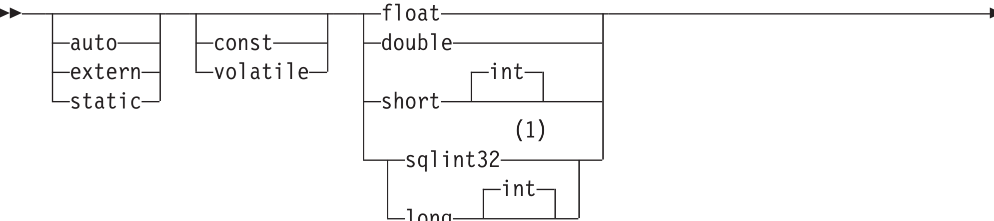
Declaring Host Variables in C

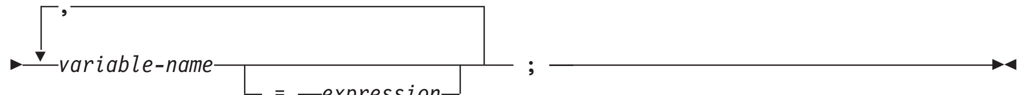
Only a subset of valid C declarations are recognized as valid host variable declarations.

Numeric Host Variables

The following figure shows the syntax for valid numeric host variable declarations.

Numeric





Notes:

1 For maximum application portability, use sqlint32 for INTEGER host variables. To use sqlint32, the header file sqlsystm.h must be included.

Character Host Variables (excluding CLOB)

There are three valid non-LOB forms for character host variables:

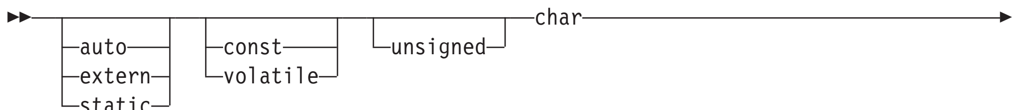
Single-character form

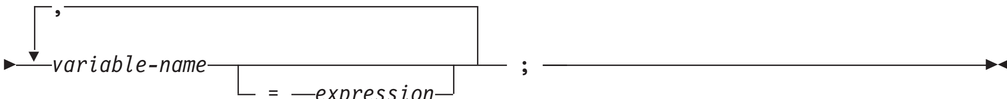
NUL-terminated character form

VARCHAR structured form

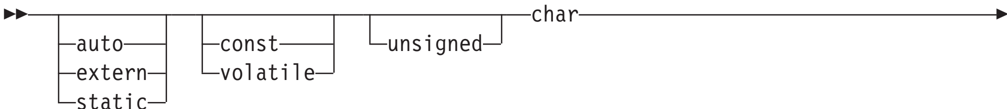
All character types are treated as unsigned.

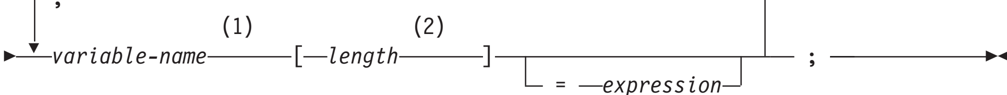
Single-Character Form





NUL-Terminated Character Form



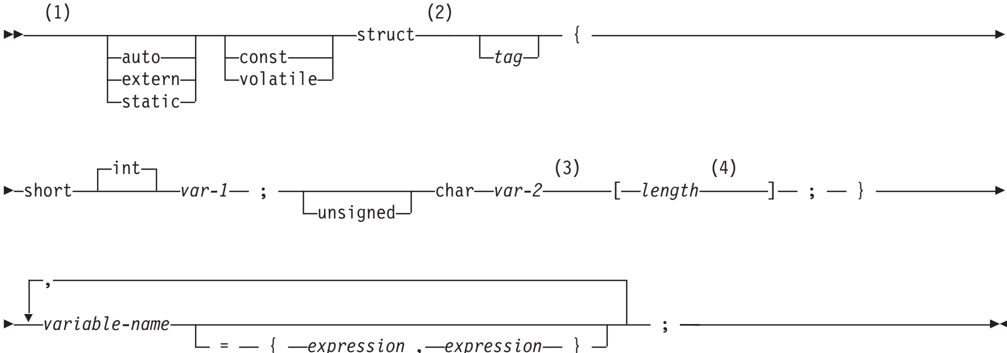


Notes:

1 On input, the string contained by the variable must be NUL-terminated. On output, the string will be NUL-terminated. [130]

2 *length* must be an integer constant greater than 1 and no greater than the maximum length of VARCHAR+1. See Table 39 on page 510 for more information.

VARCHAR Structured Form



Notes:

1 The VARCHAR structured form should be used for bit data that may contain the NULL character. The VARCHAR structured form will not be ended using the NUL-terminator.

2 The struct tag can be used to define other data areas, but these cannot be used as host variables.

3 *var-1* and *var-2* must be simple variable references and cannot be used as host variables.

4 *length* must be an integer constant that is greater than 0 and not greater than the maximum length of VARCHAR. See Table 39 on page 510 for more information.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable vstring */
struct VARCHAR
{
  short len;
  char s[10];
} vstring;

/* invalid declaration of host variable wstring */
struct VARCHAR wstring;
```

Graphic Host Variables (excluding DBCLOB)

There are three valid non-LOB forms for graphic host variables:

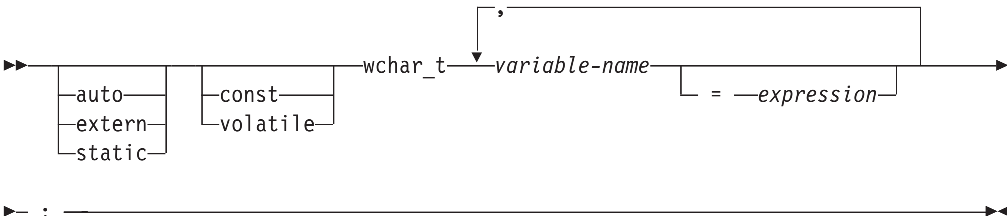
Single-graphic form

NUL-terminated graphic form

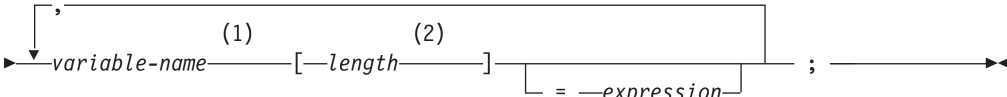
VARGRAPHIC structured form

Single-Graphic Form

130. In DB2 UDB for iSeries and DB2 UDB for UWO, a program preparation option must be used if the string will be NUL-terminated when the host variable is large enough to contain the result, but not large enough to contain the NUL-terminator. The program preparation option must also be specified for the database manager to verify that NUL-terminated input host variables contain a NUL.



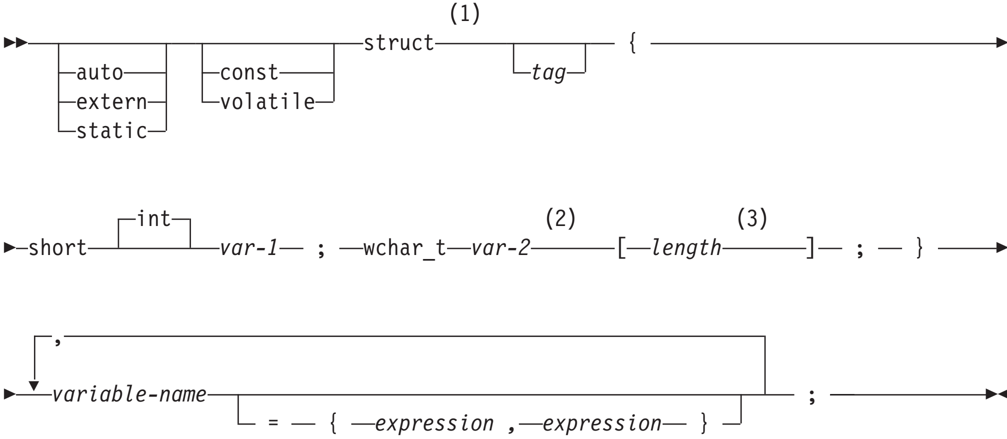
NUL-Terminated Graphic Form



Notes:

1. On input, the string contained by the variable must be NUL-terminated. On output, the string will be NUL-terminated.[130]
2. *length* must be an integer constant that is greater than 1 and not greater than the maximum length of VARGRAPHIC+1. See Table 39 on page 510 for more information.

VARGRAPHIC Structured Form



Notes:

1. The struct tag can be used to define other data areas, but these cannot be used as host variables.
2. *var-1* and *var-2* must be simple variable references and cannot be used as host variables.

3 *length* must be an integer constant that is greater than 0 and no greater than the maximum length of VARGRAPHIC. See Table 39 on page 510 for more information.

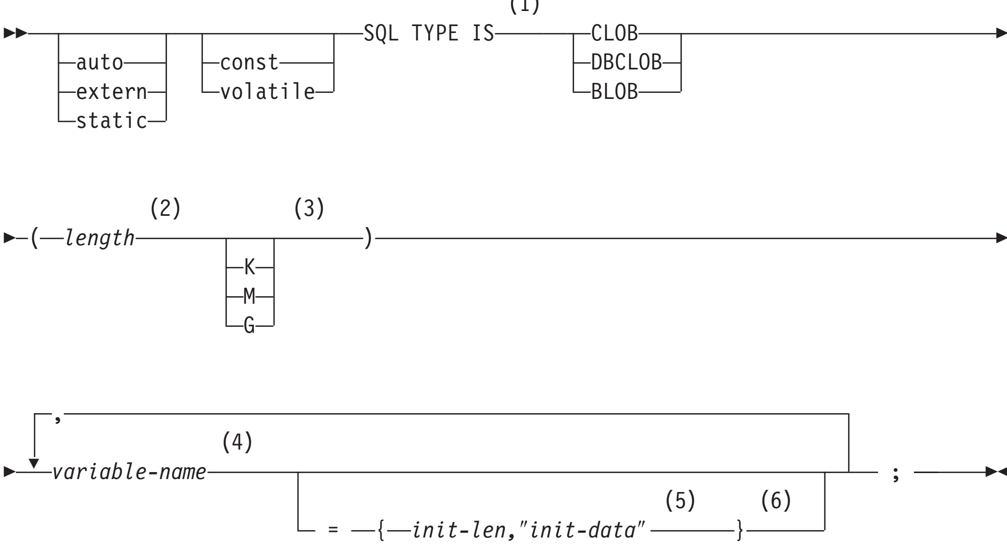
Example:

```
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable vstring */
struct VARGRAPH
{
  short len;
  wchar_t s[10];
} vstring;

/* invalid declaration of host variable wstring */
struct VARGRAPH wstring;
```

LOB Host Variables

C does not have variables that correspond to the SQL data types for LOBs (large objects). To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source.

LOB Host Variable



Notes:

1 SQL TYPE IS, CLOB, DBCLOB, BLOB, K, M, G can be in mixed case.

2 *length* must be an integer constant that is greater than 0 and no greater than the maximum length of CLOB. See Table 39 on page 510 for more information. The maximum value for *length* is further restricted if K, M or G is specified or if DBCLOB is specified.

3 K multiplies *length* by 1024. M multiplies *length* by 1 048 576. G multiplies *length* by 1 073 741 824.

4 The precompiler generates a structure tag which can be used to cast to the host variable's type.

5 The initialization length, *init-len*, must be an integer constant (that is, it

cannot include K, M, or G) that is greater than 0 and not greater than the maximum length of a character constant. See Table 39 on page 510 for more information.

6 If the LOB is not initialized within the declaration, then no initialization will be done within the precompiler generated code. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

Examples: *Example 1:* The following declaration:

```
SQL TYPE IS CLOB(128K) var1, var2 = {10, "data2data2"};
```

Results in the effective generation of the following structure:

```
struct var1_t
{
  unsigned long  length;
  char           data[131072];
} var1, var2 = {10, "data2data2"};
```

Example 2: The following declaration:

```
SQL TYPE IS DBCLOB(128K) my_dbclob;
```

Results in the effective generation of the following structure:

```
struct my_dbclob_t {
  unsigned long  length;
  wchar_t        data[131072];
} my_dbclob;
```

Example 3: The following declaration:

```
SQL TYPE IS BLOB(128K) my_blob;
```

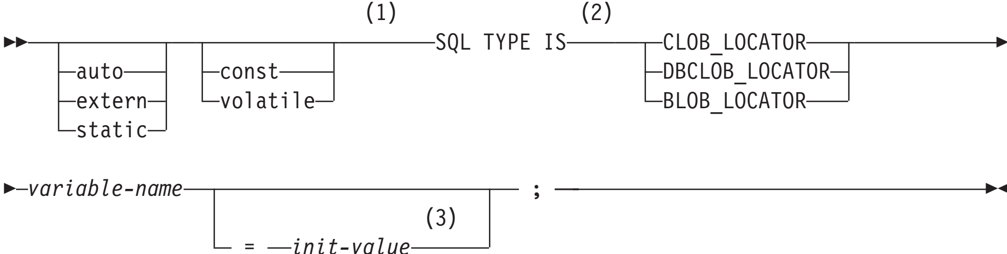
Results in the effective generation of the following structure:

```
struct my_blob_t {
  unsigned long  length;
  char           data[131072];
} my_blob;
```

LOB Locator

The following shows the syntax for declaring large object locator host variables in C.

LOB Locator



Notes:

1 Pointers to LOB Locators can be declared, with the same rules and restrictions as for pointers to other host variable types.

2 SQL TYPE IS, CLOB_LOCATOR, DBCLOB_LOCATOR, BLOB_LOCATOR can be in mixed case.

3 *init-value* permits the initialization of pointer locator variables. Other types of initialization will have no meaning.

Example: The following declaration:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

Results in the effective generation of the following:

```
unsigned long my_locator;
```

DBCLOB and BLOB locators have similar syntax.

Indicator Variables in C

An indicator variable is a two-byte integer (short int). On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to indicate that a null value should be assigned.

See "References to Host Variables" on page 85 for more information on the use of indicator variables.

Indicator variables are declared in the same way as host variables, and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

Example: Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
                               :Day :DayInd,
                               :Bgn :BgnInd,
                               :End :EndInd;
```

Variables can be declared as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char  ClsCd[8];
char  Bgn[9];
char  End[9];
short Day, DayInd, BgnInd, EndInd;
EXEC SQL END DECLARE SECTION;
```

Declaring Host Structures in C

Host structures can be defined in C programs. A host structure contains an ordered group of elementary C variables. It can have a maximum of two levels, even though the host structure might itself occur within a multilevel structure. The one exception is the declaration of a varying-length string, which requires another structure and hence one more level. When the host structure occurs within a multilevel structure, it must be the deepest level of the nested structure. The following is an example of a host structure.

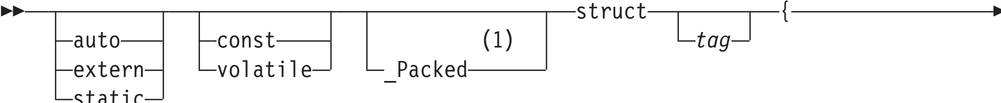
```
struct
{
  char  c1[3];
  struct
  {
    short len;
```

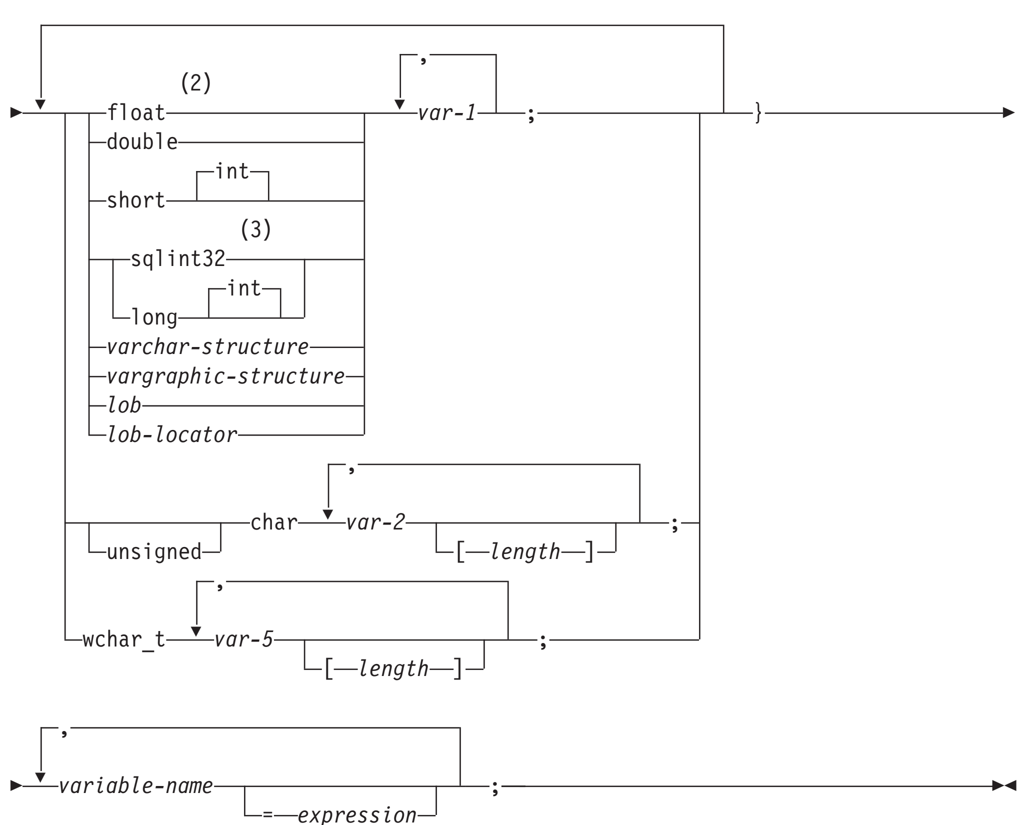
```
    char  data[5];
  } c2;
  char  c3[2];
} target;
```

In this example, *target* is the name of a host structure consisting of the *c1*, *c2*, and *c3* fields. *c1* and *c3* are character arrays, and *c2* is the host variable equivalent to the VARCHAR structured form.

The following shows the syntax for valid host structures:

Host Structures





Notes:

1 _Packed must not be used in C++. Instead, specify #pragma pack(1) prior to the declaration and #pragma pack() after the declaration.

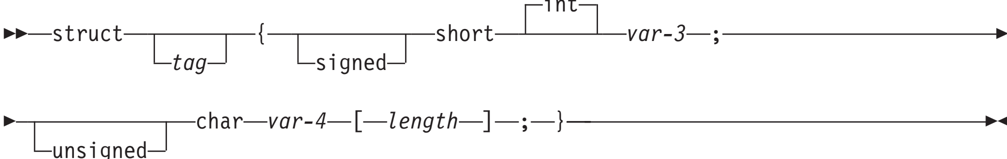
```
#pragma pack(1)
struct
{
  short myshort;
```

```
  long  mylong;
  char  mychar[5];
} a_st;
#pragma pack()
```

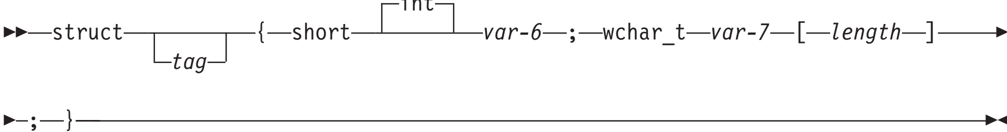
2 For details on declaring numeric, character, and graphic host variables, see the notes under numeric, character and graphic host variables.

3 To use sqlint32, the header file sqlsystm.h must be included.

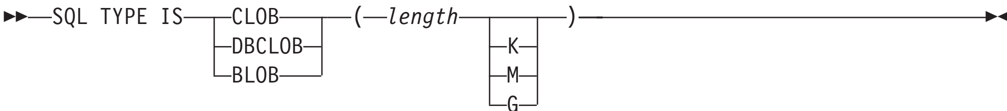
varchar-structure



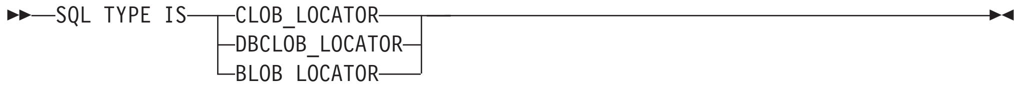
vargraphic-structure



lob



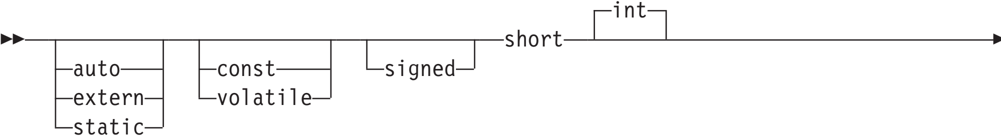
lob-locator

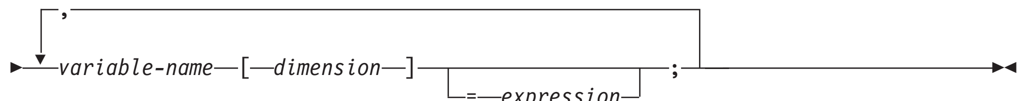


Host Structure Indicator Array

The following figure shows the valid syntax for host structure indicator array declarations.

Host Structure Indicator Array





Note: Dimension must be an integer constant between 1 and 32767.

Using Pointer Data Types in C

A host variable declared in C using pointer notation must be used as a pointer to an SQL descriptor area. A *descriptor-name* can be specified in the CALL, DESCRIBE, EXECUTE, FETCH, and OPEN statements. For example, *descriptor-name* could be declared as:

```
sqlda *outsqlda;
```

and used in a statement as follows:

```
EXEC SQL DESCRIBE STMT1 INTO DESCRIPTOR :*outsqlda;
```

Determining Equivalent SQL and C Data Types

The base SQLTYPE and SQLLEN of host variables are determined according to the following table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

Table 66. C Declarations Mapped to Typical SQL Data Types

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
short int	500/501	2	SMALLINT
long int	496/497	4	INTEGER
float	480/481	4	FLOAT (single precision)
double	480/481	8	FLOAT (double precision)
single-character form	452/453	1	CHAR(1)
NUL-terminated character form	460/461	length	VARCHAR (length - 1)
VARCHAR structured form	448/449, 456/457	length	VARCHAR (length)
single-graphic form	468/469	1	GRAPHIC(1)
NUL-terminated graphic form (wchar_t)	400/401	length	VARGRAPHIC (length - 1)
VARGRAPHIC structured form	464/465, 472/473	length	VARGRAPHIC (length)
SQLTYPE IS CLOB	408/409	length	CLOB (length)
SQLTYPE IS DBCLOB	412/413	length	DBCLOB (length)
SQLTYPE IS BLOB	404/405	length	BLOB (length)
SQLTYPE IS CLOB_LOCATOR	964/965	4	CLOB locator [131]
SQLTYPE IS DBCLOB_LOCATOR	968/969	4	DBCLOB locator [131]
SQLTYPE IS BLOB_LOCATOR	960/961	4	BLOB locator [131]

The following table can be used to determine the C data type that is equivalent to a given SQL data type.

Table 67. SQL Data Types Mapped to Typical C Declarations

SQL Data Type	C Data Type	Notes
SMALLINT	short int	
INTEGER	long int	
DECIMAL(p,s) or NUMERIC(p,s)	no exact equivalent	Use double.
FLOAT (single precision)	float	
FLOAT (double precision)	double	
CHAR(1)	single-character form	
CHAR(n)	no exact equivalent	If n>1, use NUL-terminated character form
VARCHAR(n)	NUL-terminated character form	Allow at least n + 1 to accommodate the NUL-terminator. If data can contain character NULs (\0), use VARCHAR structured form. *n* is a positive integer. The maximum value of *n* is 32 672. See Table 39 on page 510 for more information.
	VARCHAR structured form	*n* is a positive integer. The maximum value of *n* is 32 672. See Table 39 on page 510 for more information.
CLOB(n)	SQL TYPE IS CLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 39 on page 510 for more information.
GRAPHIC(1)	single-graphic form	
GRAPHIC(n)	no exact equivalent	If n>1, use NUL-terminated graphic form
VARGRAPHIC(n)	NUL-terminated graphic form	Allow at least n + 1 to accommodate the NUL-terminator. If data can contain graphic NUL values (/0/0), use VARGRAPHIC structured form. *n* is a positive integer. The maximum value of *n* is 16 336. See Table 39 on page 510 for more information.
	VARGRAPHIC structured from	*n* is a positive integer. The maximum value of *n* is 16 336. See Table 39 on page 510 for more information.
DBCLOB(n)	SQL TYPE IS DBCLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 39 on page 510 for more information.
BLOB(n)	SQL TYPE IS BLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647. See Table 39 on page 510 for more information.

131. Do not use this data type as a column type.

Table 67. SQL Data Types Mapped to Typical C Declarations (continued)

SQL Data Type	C Data Type	Notes
DATE	NUL-terminated character form	Allow at least 11 characters to accommodate the NUL-terminator.
	VARCHAR structured form	Allow at least 10 characters.
TIME	NUL-terminated character form	Allow at least 7 characters (9 to include seconds) to accommodate the NUL-terminator.
	VARCHAR structured form	Allow at least 6 characters; 8 to include seconds.
TIMESTAMP	NUL-terminated character form	Allow at least 20 characters (27 to include microseconds at full precision) to accommodate the NUL-terminator.
	VARCHAR structured form	Allow at least 19 characters; 26 to include microseconds at full precision.

Appendix I. Coding SQL Statements in COBOL Applications

This section describes the programming techniques that are unique to coding SQL statements within a COBOL program.

Defining the SQL Communications Area in COBOL

A COBOL program that contains SQL statements must include one or both of the following:

- An SQLCODE variable[133] declared as PICTURE S9(9) BINARY, PICTURE S9(9) COMP-4, or PICTURE S9(9) COMP [132]
- An SQLSTATE variable[133] declared as PICTURE X(5)

or,

- An SQLCA (which contains an SQLCODE and SQLSTATE variable).

The SQLCODE and SQLSTATE values are set by the database manager after each SQL statement is executed. An application can check the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful. See Appendix E, "SQLSTATE Values—Common Return Codes" on page 539 for more information.

The SQLCA can be coded in a COBOL program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

The SQLCA must not be defined within an SQL declare section. See Appendix C, "SQL Communication Area (SQLCA)" on page 525 and "INCLUDE" on page 421 for more information.

The SQLSTATE, SQLCODE, and SQLCA variables must appear in the WORKING-STORAGE SECTION or LINKAGE SECTION of the program and can be placed wherever a record description entry can be specified in those sections.

Defining SQL Descriptor Areas in COBOL

The following statements require an SQLDA:

EXECUTE...USING DESCRIPTOR *descriptor-name*

FETCH...USING DESCRIPTOR *descriptor-name*

OPEN...USING DESCRIPTOR *descriptor-name*

DESCRIBE *statement-name* INTO *descriptor-name*

PREPARE *statement-name* INTO *descriptor-name*

CALL...USING DESCRIPTOR *descriptor-name*

132. In DB2 UDB for UWO, the SQLCODE variable must be declared as COMP-5.

133. In DB2 UDB for z/OS and OS/390, the STDSQL(YES) option must be in effect to declare the SQLSTATE and SQLCODE variables. In DB2 UDB for UWO, the LANGLEVEL SQL92E option must be used to declare the SQLSTATE and SQLCODE variables.

© Copyright IBM Corp. 1982, 2002

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name.

The SQLDA can be coded in a COBOL program either directly or by using the SQL INCLUDE statement. The SQLDA must not be defined within an SQL declare section. See Appendix D, “SQL Descriptor Area (SQLDA)” on page 529 and “INCLUDE” on page 421 for more information. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLDA END-EXEC.
```

SQLDA declarations must appear in the WORKING-STORAGE SECTION or LINKAGE SECTION of the program and can be placed wherever a record description entry can be specified in those sections.

Embedding SQL Statements in COBOL

SQL statements can be coded in COBOL program sections as follows:

SQL Statement	Program Section
BEGIN DECLARE SECTION END DECLARE SECTION	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE SQLCA INCLUDE SQLDA	WORKING-STORAGE SECTION
DECLARE CURSOR INCLUDE name	DATA DIVISION or PROCEDURE DIVISION
Other	PROCEDURE DIVISION

SQL statements must not be coded in COBOL programs with more than one PROCEDURE DIVISION.

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If the SQL statement appears between two COBOL statements, the period is optional and might not be appropriate. The EXEC SQL keywords must appear all on one line, but the remainder of the statement can appear on the next and subsequent lines.

For example, an UPDATE statement coded in a COBOL program might be coded as follows:

```
EXEC SQL
   UPDATE DEPARTMENT
   SET MGRNO = :MGR-NUM
   WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Comments

In addition to SQL comments (--), COBOL comment lines (* in column 7) can be included within embedded SQL statements, except between the keywords EXEC and SQL. [134]

134. In DB2 UDB for z/OS and OS/390, the STDSQL(YES) option must be in effect to use SQL comments.

Continuation for SQL statements

The line continuation rules for SQL statements are the same as those for other COBOL statements, except that EXEC SQL must be specified within one line.

If a string constant is continued from one line to the next, the first nonblank character in the next line must be either an apostrophe or a quotation mark. In DB2 UDB for UWO this character must be an apostrophe. Identifiers that are not delimited cannot be continued. If a delimited identifier is continued from one line to the next, the first nonblank character in the next line must be either an apostrophe or a quotation mark. In DB2 UDB for UWO this character must be a quotation mark.

Cursors

The DECLARE CURSOR statement must precede all statements that explicitly refer to the cursor by name.

Including Code

SQL statements or COBOL host variable declaration statements can be included by embedding the following SQL statement at the point in the source code where the statements are to be embedded:

```
EXEC SQL INCLUDE name END-EXEC.
```

COBOL COPY statements cannot be used to include SQL statements or declarations of COBOL variables that are referenced in SQL statements.

Margins

SQL statements must be coded in columns 12 through 72.

Names

Any valid COBOL variable name can be used for a host variable, as long as it:
- does not contain DBCS characters
- does not begin with 'DB2', 'DSN', 'RDI', or 'SQL' in any combination of uppercase or lowercase letters (these names are reserved for the database manager).

It is recommended that FILLER not be used as a variable name. All fields within a COBOL structure should be named to avoid unexpected results from using structures that contain FILLER.

Access plan names must not start with ’DSN’. External entry names must not start with ’DSN’, ’RDI’, or ’SQL’.

For information on the length of a host identifier, see Table 37 on page 509.

Statement Labels

Executable SQL statements in the PROCEDURE DIVISION can be preceded with a paragraph name.

Handling SQL Errors and Warnings in COBOL

The SQL WHENEVER statement tests the result of every SQL statement within its scope for an error or warning condition. The target for the GOTO clause in an SQL WHENEVER statement must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

The stand-alone SQLSTATE and SQLCODE or information in the SQLCA can also be used in the detection or further handling of error and warning conditions. See Appendix C, “SQL Communication Area (SQLCA)” on page 525 for more information.

Using Host Variables in COBOL

A host variable used in an SQL statement must be explicitly declared prior to the first use of the host variable in an SQL statement.

The COBOL statements that are used to define the host variables must be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement.

All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.

Host variables must not be arrays or array elements unless they are used to represent indicator arrays or indicator variables. Host variables must not be records or elements.

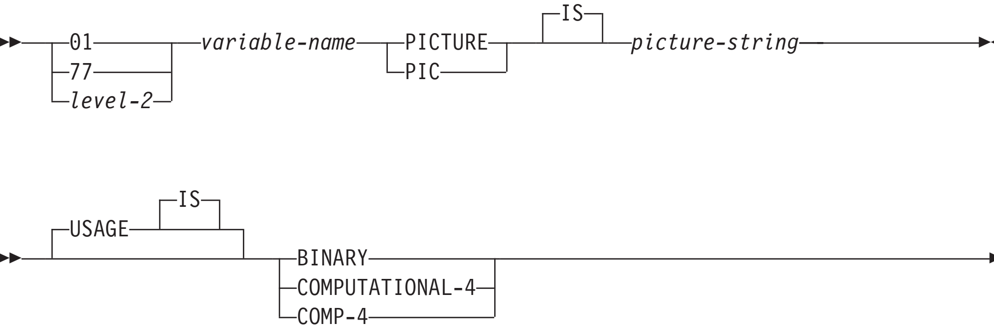
Declaring Host Variables in COBOL

Only a subset of valid COBOL declarations are recognized as valid host variable declarations.

Numeric Host Variables

The following figures show the syntax for valid integer host variable declarations.

INTEGER and SMALLINT



```
►─┬───────────────────────────────┬─ . ─────────────────────►◄
  │          ┌─IS─┐               │
  └─VALUE────┴────┴──numeric-constant─┘
```

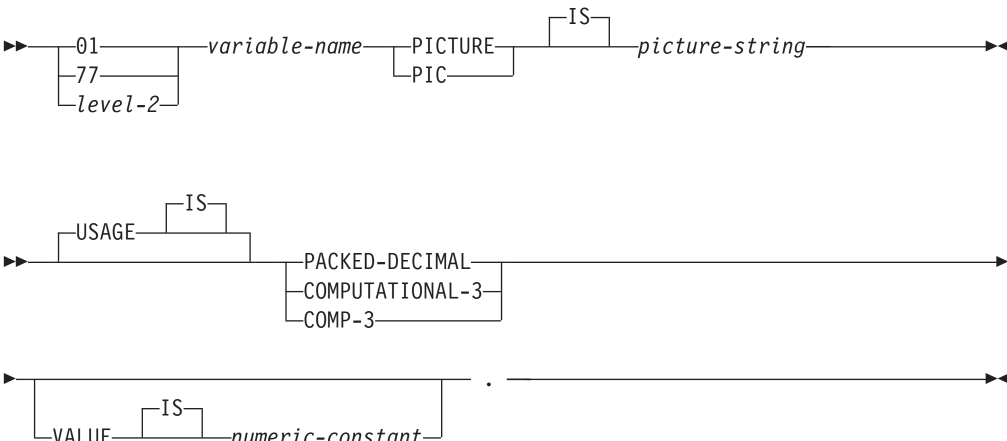
Notes:

- BINARY, COMPUTATIONAL-4, COMP-4, are equivalent. COMPUTATIONAL-4 and COMP-4 are IBM extensions that are not supported in ISO/ANS COBOL. The *picture-string* associated with these types must be either S9(4), S9999, S9(9) or S999999999.

  In DB2 UDB for UWO, these declarations are not supported; COMPUTATIONAL-5 or COMP-5 must be used instead.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid decimal host variable declarations.

DECIMAL

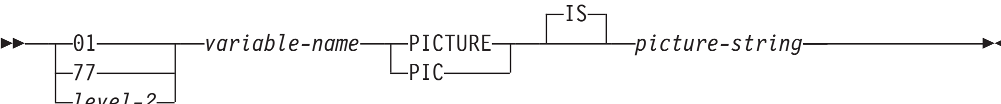


Notes:

- PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. COMPUTATIONAL-3 and COMP-3 are IBM extensions that are not supported in ISO/ANS COBOL. The *picture-string* associated with these types must have the form S9(i)V9(d) (or S9...9V9...9, with *i* and *d* instances of 9). ISO/ANSI COBOL restricts i + d to be less than or equal to 18.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid numeric host variable declarations.

NUMERIC



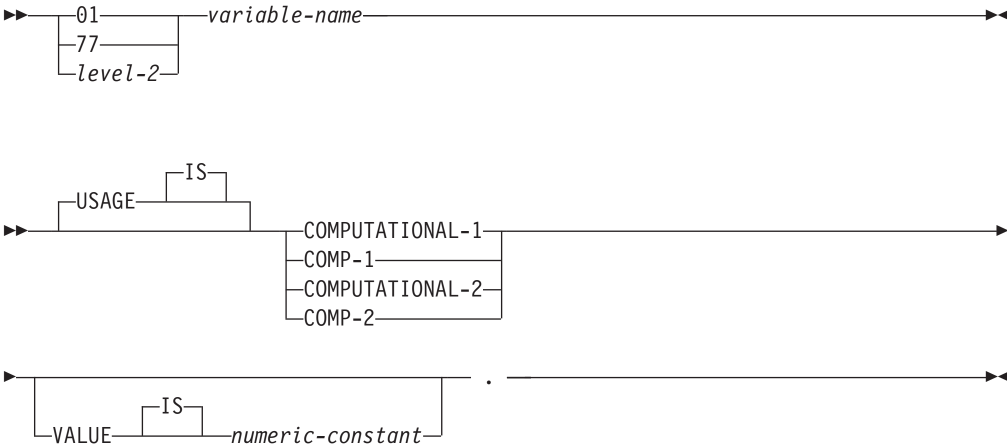
```
              ┌─IS─┐
   ┌─USAGE────┴────┴─┐
►►─┴─────────────────┴─DISPLAY SIGN LEADING SEPARATE──────────────────────►

►─┬─────────────────────────────────────┬─ . ─────────────────────────────►◄
  └─VALUE──┬────┬──numeric-constant─────┘
           └─IS─┘
```

Notes:

- The picture-string associated with SIGN LEADING SEPARATE must have the form S9(i)V9(d) (or S9...9V9...9, with *i* and *d* instances of 9). ISO/ANSI COBOL restricts i + d to be less than or equal to 18. In DB2 UDB for UWO, SIGN LEADING SEPARATE is not supported.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid floating-point host variable declarations.

Floating Point

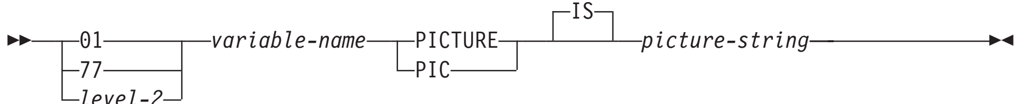


Notes:

- COMPUTATIONAL-1 and COMP-1 are equivalent. COMPUTATIONAL-2 and COMP-2 are equivalent.
- *level-2* indicates a COBOL level between 2 and 48.

The following figure shows the syntax for other valid numeric host variable declarations.

Other Numeric



```
               ┌─IS─┐
    ┌─USAGE────┴────┴─┐
►►──┴─────────────────┴──┬─COMPUTATIONAL─┬──┬──────────────────────────────────┬──►
                         └─COMP──────────┘  │         ┌─IS─┐                   │
                                            └─VALUE───┴────┴─numeric-constant──┘

►── . ─────────────────────────────────────────────────────────────────────────►◄
```

Notes:

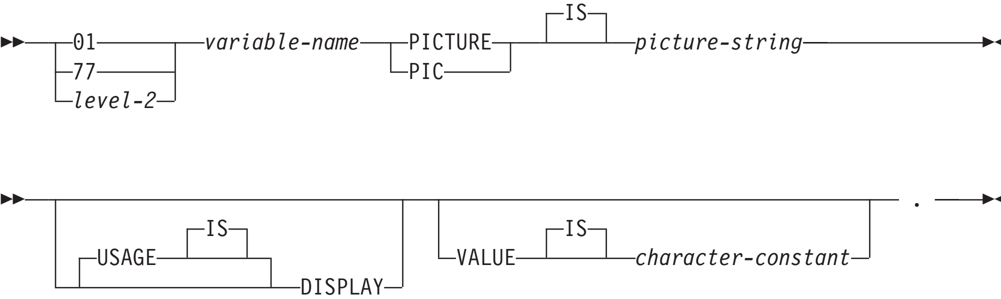
- COMPUTATIONAL and COMP are equivalent. The picture strings associated with these and the data types they represent are product-specific. Therefore, COMP and COMPUTATIONAL should not be used in a portable application.
- *level-2* indicates a COBOL level between 2 and 48.

Character Host Variables (excluding CLOB)

There are two valid non-LOB forms of character host variables:

- Fixed-Length Strings
- Varying-Length Strings

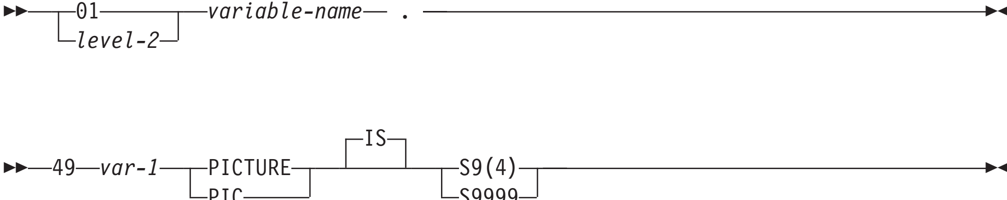
Fixed-Length Character Strings

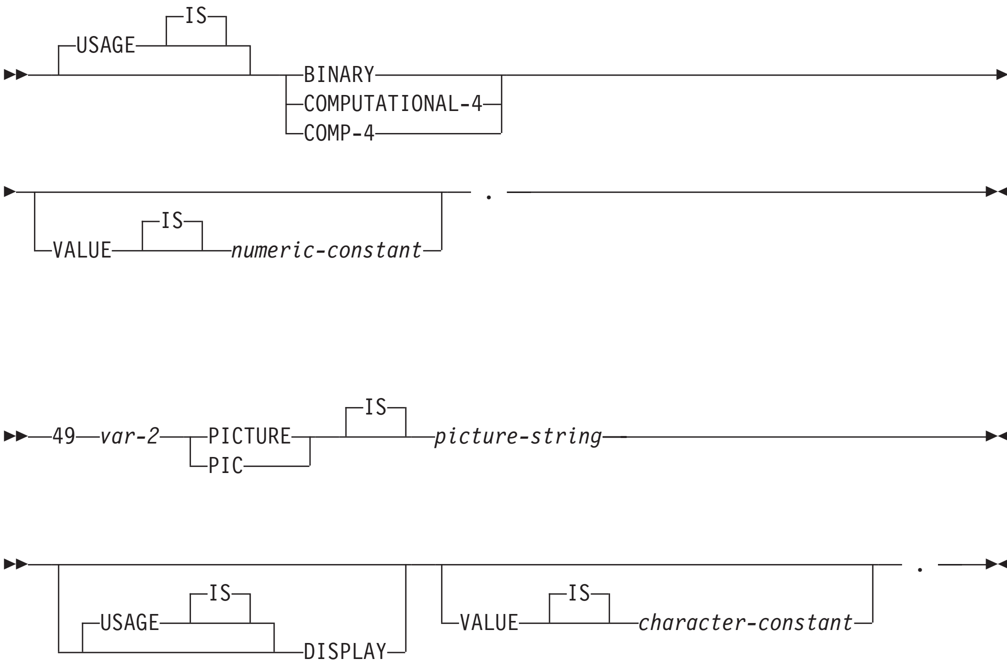


Notes:

- The *picture-string* associated with this form must be X(*m*) (or XXX...X, with *m* instances of X). *m* must be no greater than the maximum length of CHAR. See Table 39 on page 510 for more information.
- *level-2* indicates a COBOL level between 2 and 48.

Varying-Length Character Strings





Notes:

- The *picture-string* associated with this form must be X(m) (or XXX...X, with *m* instances of X). *m* can be no greater than the maximum length of VARCHAR. See Table 39 on page 510 for more information.

  G In DB2 UDB for UWO, COMP(5) must be used in place of COMP(4).

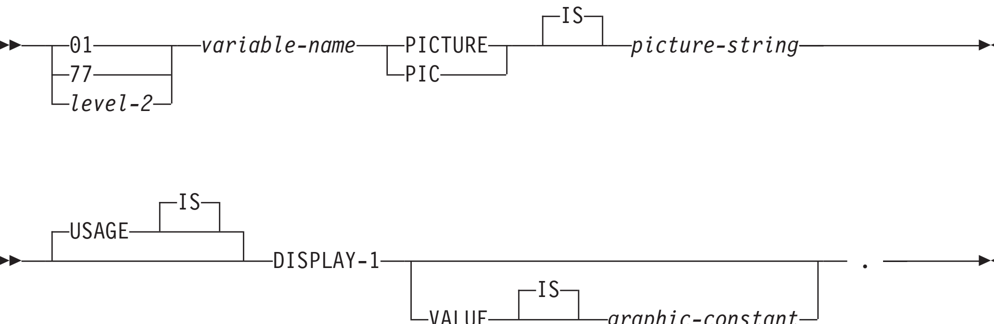
  Note that the database manager will use the full size of the S9(4) variable even though ISO/ANSI COBOL only recognizes values up to 9999. This can cause data truncation errors when COBOL statements are being executed and may effectively limit the maximum length of variable-length character strings to 9999.
- *var-1* and *var-2* cannot be used as host variables.
- *level-2* indicates a COBOL level between 2 and 48.

Graphic Host Variables (excluding DBCLOB)

There are two valid non-LOB forms for graphic host variables:

- Fixed-Length Strings
- Varying-Length Strings

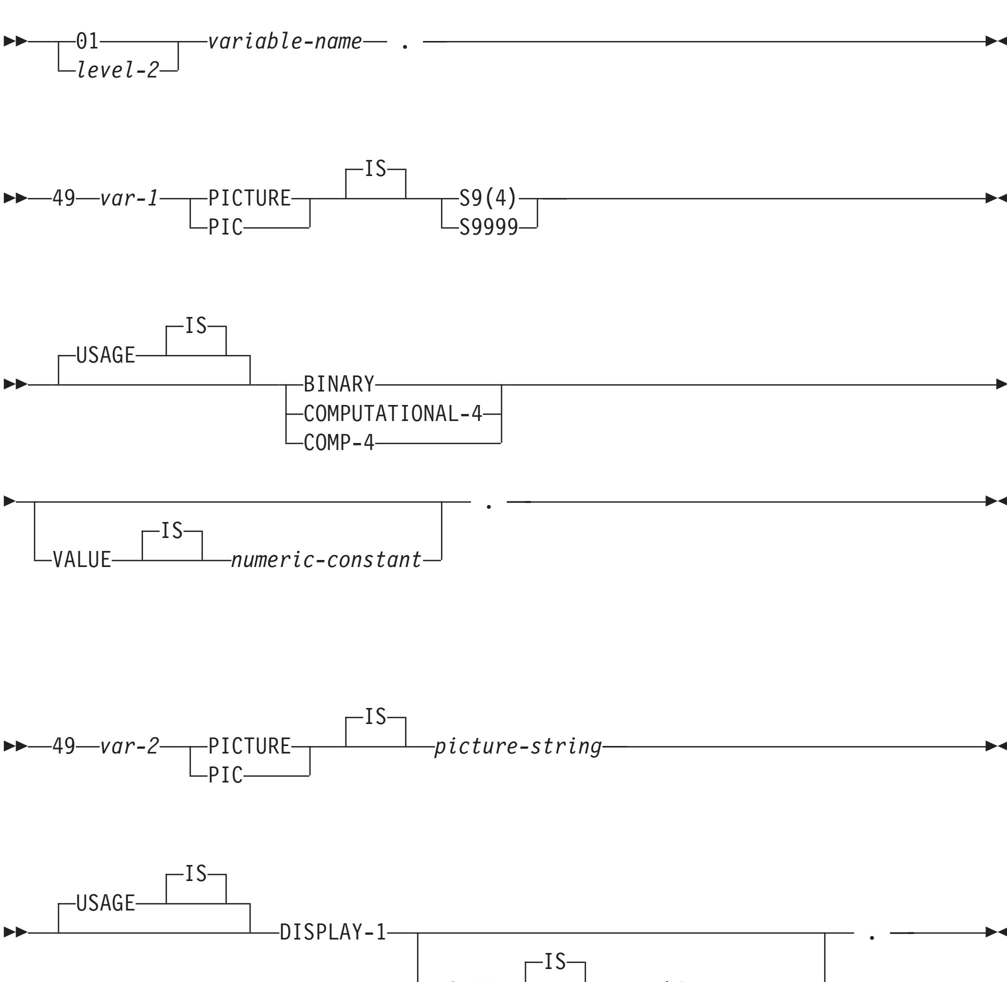
Fixed-Length Graphic Strings



Notes:

- The *picture-string* associated with this form must be G(*m*) (or GGG...G, with *m* instances of G. *m* must be no greater than the maximum length of GRAPHIC. See Table 39 on page 510 for more information.
- *level-2* indicates a COBOL level between 2 and 48.

Varying-Length Graphic Strings



Notes:

- The *picture-string* associated with this form must be G(*m*) (or GGG...G, with *m* instances of G). *m* must be no greater than the maximum length of VARGRAPHIC. See Table 39 on page 510 for more information.

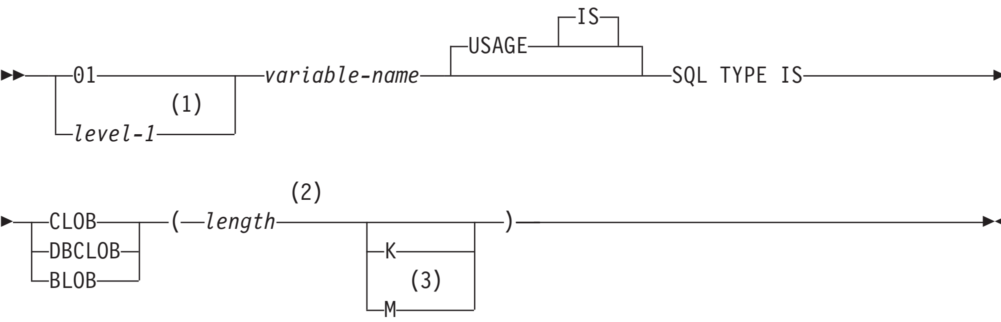
  Note that the database manager will use the full size of the S9(4) variable even though ISO/ANSI COBOL only recognizes values up to 9999. This can cause data truncation errors when COBOL statements are being executed and may effectively limit the maximum length of variable-length graphic strings to 9999.

  G In DB2 UDB for UWO, COMP-5 must be used in place of COMP-4.
- *var*-1 and *var*-2 cannot be used as host variables.
- *level-2* indicates a COBOL level between 2 and 48.

LOB Host Variables

COBOL does not have variables that correspond to the SQL data types for LOBs (large objects). To define host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source.

LOB Host Variable



Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. *length* must be an integer constant that is greater than 0 and no greater than the maximum length of CLOB. See Table 39 on page 510 for more information. The maximum value for *length* is further restricted if K or M or if DBCLOB is specified.
3. K multiplies *length* by 1024. M multiplies *length* by 1 048 576.

Examples: *Example 1:* The following declaration:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

Results in the generation of the following structure:

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

Example 2: The following declaration:

```
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

Results in the generation of the following structure:

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

Example 3: The following declaration:

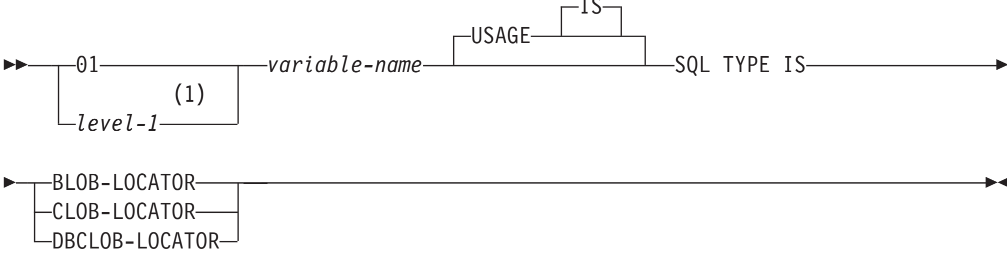
```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

Results in the generation of the following structure:

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

LOB Locators

LOB Locator



Notes:

1 *level-1* indicates a COBOL level between 2 and 48.

Example: The following declaration (other LOB locator types are similar):

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

Indicator Variables in COBOL

An indicator variable is a two-byte integer (PIC S9(4) USAGE BINARY). On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to indicate that a null value should be assigned.

See “References to Host Variables” on page 85 for more information on the use of indicator variables.

Indicator variables are declared in the same way as host variables, and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

Example: Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO
                               :DAY-VAR :DAY-IND,
                               :BGN-VAR :BGN-IND,
                               :END-VAR :END-IND
END-EXEC.
```

Variables can be declared as follows:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 CLS-CD    PIC X(7).
77 DAY-VAR   PIC S9(4) BINARY.
77 BGN-VAR   PIC X(8).
77 END-VAR   PIC X(8).
77 DAY-IND   PIC S9(4) BINARY.
77 BGN-IND   PIC S9(4) BINARY.
77 END-IND   PIC S9(4) BINARY.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Declaring Host Structures in COBOL

A COBOL host structure is a named set of host variables that is defined in the program's WORKING-STORAGE SECTION or LINKAGE SECTION. COBOL host structures have a maximum of two levels, even though the host structure might occur within a multilevel structure. One exception is the declaration of a varying-length character string, which must be level 49.

A host structure name can be a group name whose subordinate levels name elementary data items. In the following example, B is the name of a host structure consisting of the elementary items C1 and C2.

```
01 A
  02 B
    03 C1 PICTURE ...
    03 C2 PICTURE ...
```

When writing an SQL statement using a qualified host variable name (for example, to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, specify B.C1 rather than C1 OF B or C1 IN B.

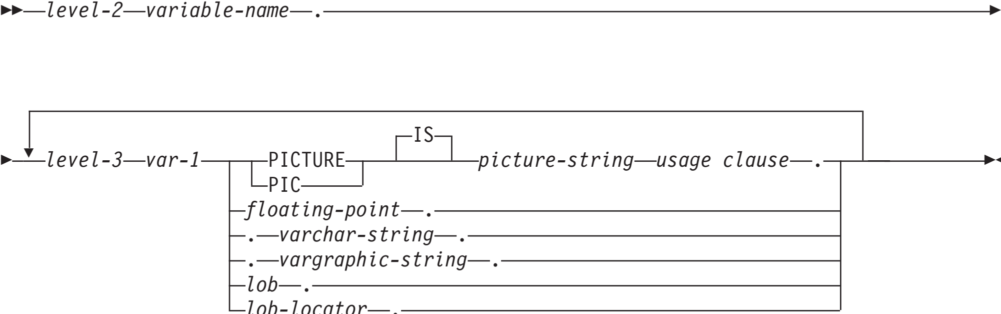
A host structure is considered complete if any of the following items are found:

- A COBOL item that must begin in area A
- Any SQL statement (except SQL INCLUDE).
- Any SQL statement within an included member.

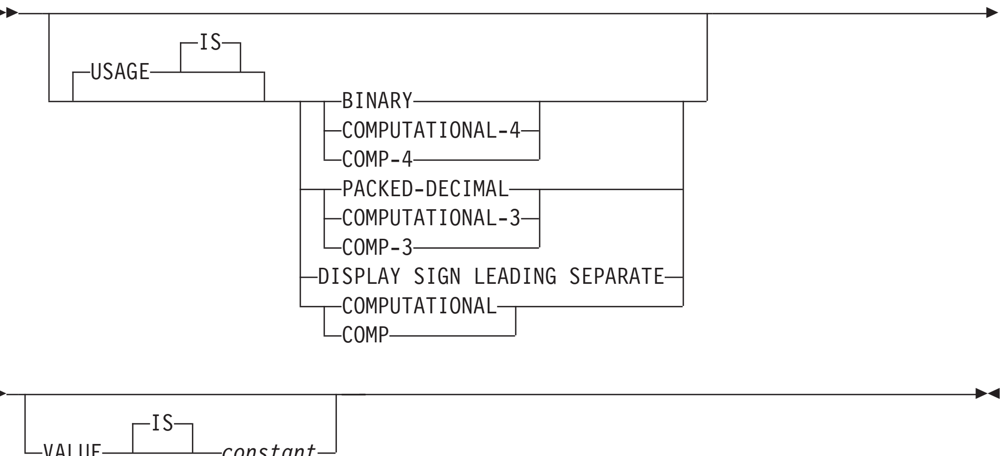
All fields within a COBOL structure should be named to avoid unexpected results that might result from using structures that contain FILLER.

The following figure shows the syntax for valid host structures.

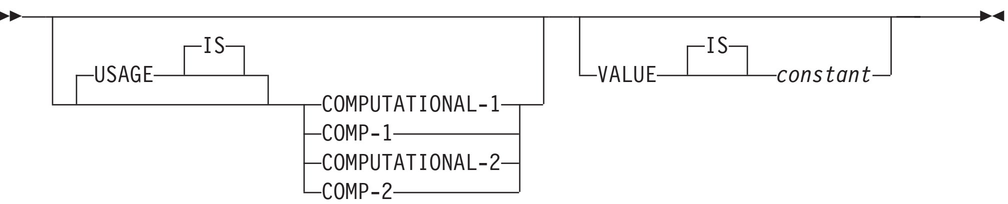
Host Structures



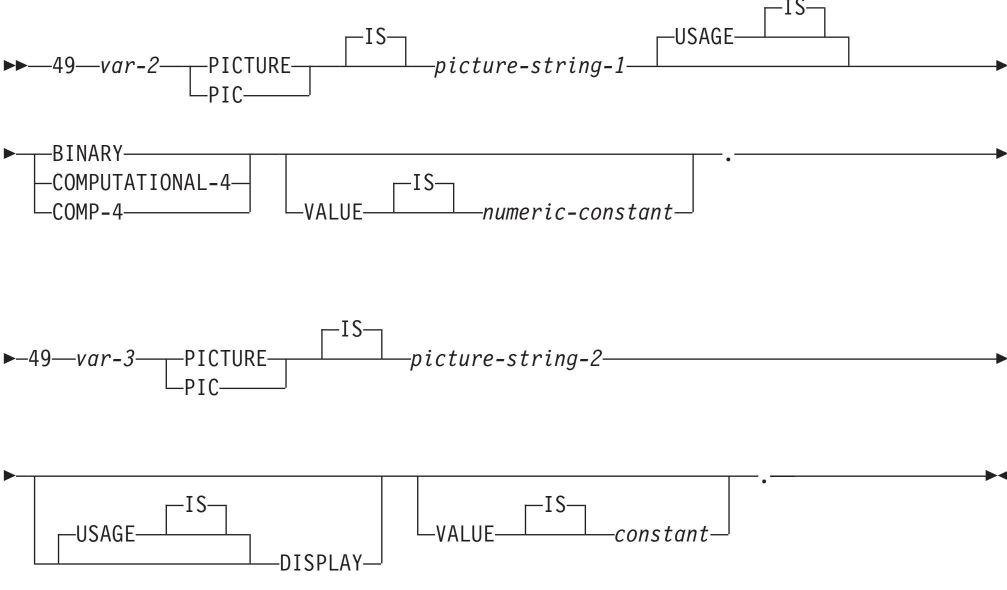
exact numeric:



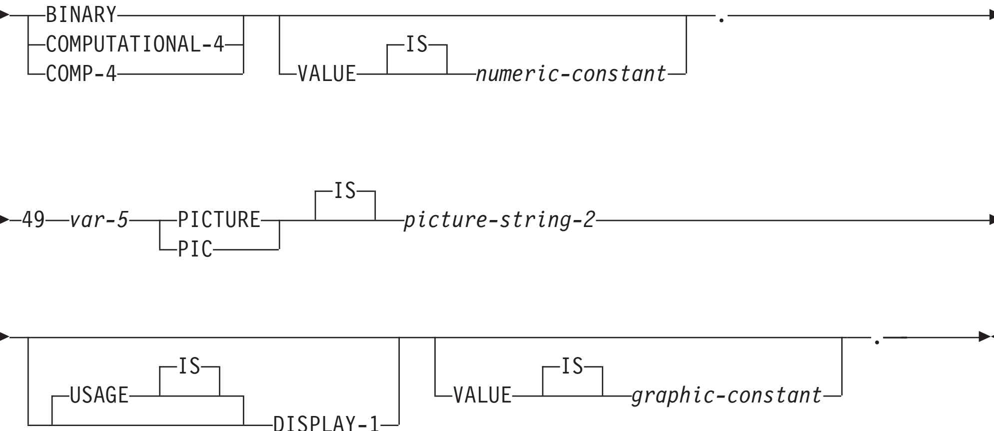
floating-point:



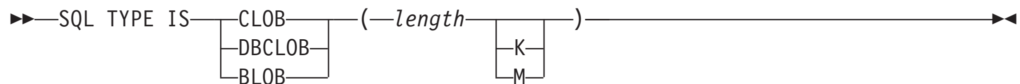
varchar-string



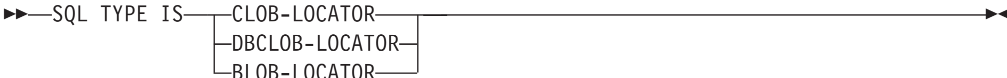
vargraphic-string



lob



lob-locator



Notes:

- *level-2* indicates a COBOL level between 1 and 47.
- *level-3* indicates a COBOL level between 2 and 48.

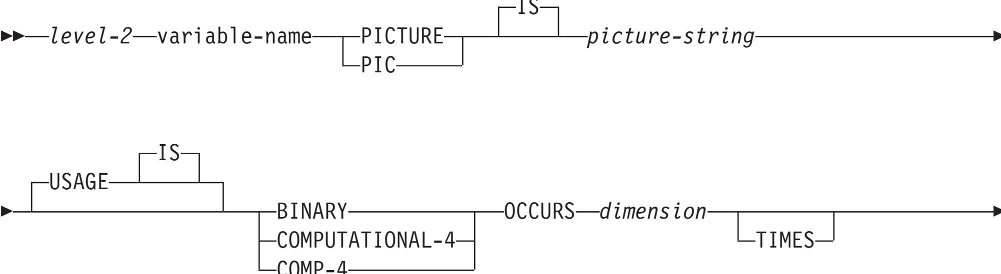
G

- In DB2 UDB for UWO, COMP-5 must be used in place of COMP-4.

Host Structure Indicator Array

The following figure shows the syntax for valid indicator array declarations.

Host Structure Indicator Array



```
►─┬──────────────────────────┬─.─────────────────────────────────►◄
  │          ┌─IS─┐          │
  └─VALUE────┴────┴─constant─┘
```

Notes:

1. Dimension must be an integer between 1 and 32767.
2. *level-2* must be an integer between 2 and 48.
3. BINARY, COMPUTATIONAL-4, and COMP-4 are equivalent. COMPUTATIONAL-4 and COMP-4 are IBM extensions that are not supported in ISO/ANSI COBOL. The *picture-string* associated with these types must have the form S9(i) (or S9...9, with i instances of 9). i must be less than or equal to 4. In DB2 UDB for UWO, COMP-5 must be used in place of COMP-4.

Determining Equivalent SQL and COBOL Data Types

The base SQLTYPE and SQLLEN of host variables are determined according to the following table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

Table 68. COBOL Declarations Mapped to Typical SQL Data Types

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
COMP-1	480/ 481	4	FLOAT (single precision)
COMP-2	480/ 481	8	FLOAT (double precision)
S9(i)V9(d) COMP-3 or S9(i)V9(d) PACKED-DECIMAL	484/ 485	i+d in byte 1, d in byte 2	DECIMAL(i+d,d)
S9(i)V9(d) DISPLAY SIGN LEADING SEPARATE [136]	504/ 505	i+d in byte 1, d in byte 2	No exact equivalent. Use DECIMAL(i+d,d) or NUMERIC (i+d,d)
S9(4) COMP-4 [135] or S9(4) BINARY	500/ 501	2	SMALLINT
S9(9) COMP-4 [135] or S9(9) BINARY	496/ 497	4	INTEGER
Fixed-length character data	452/ 453	length	CHAR(m)
Varying-length character data	448/ 449, 456/ 457	length	VARCHAR(length)
Fixed-length graphic data	468/ 469	length	GRAPHIC(length)
Varying-length graphic data	464/ 465, 472/ 473	length	VARGRAPHIC(length)
USAGE IS SQL TYPE IS CLOB(n) n < 2147483648	408/ 409	length	CLOB(length)
USAGE IS SQL TYPE IS DBCLOB(m) m < 1073741824	412/ 413	length	DBCLOB(length)
USAGE IS SQL TYPE IS BLOB(n) n < 2147483648	404/ 405	length	BLOB(length)
SQL TYPE IS CLOB-LOCATOR	964/ 965	4	CLOB locator [137]

Table 68. COBOL Declarations Mapped to Typical SQL Data Types (continued)

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
SQL TYPE IS DBCLOB-LOCATOR	968/ 969	4	DBCLOB locator [137]
SQL TYPE IS BLOB-LOCATOR	960/ 961	4	BLOB locator [137]

The following table can be used to determine the COBOL data type that is equivalent to a given SQL data type:

Table 69. SQL Data Types Mapped to Typical COBOL Declarations

SQL Data Type	COBOL Data Type	Notes
SMALLINT	S9(4) COMP-4	
INTEGER	S9(9) COMP-4	
DECIMAL(p,s) or NUMERIC(p,s)	If p < 19: S9(p-s)V9(s) PACKED-DECIMAL or S9(p-s)V9(s) DISPLAY SIGN LEADING SEPARATE If p > 18: no exact equivalent	0<=s<=p<=18, where *s* is the scale and *p* is the precision. If *s*=0, use S9(p) or S9(p)V. If *s*=p, use SV9(s). Use COMP-2
FLOAT (single precision)	COMP-1	
FLOAT (double precision)	COMP-2	
CHAR(n)	fixed-length character string	*n* is a positive integer. The maximum value of *n* is 254. See Table 39 on page 510 for more information.
VARCHAR(n)	varying-length character string	*n* is a positive integer. The maximum value of *n* is 32 672. See Table 39 on page 510 for more information.
CLOB(n)	USAGE IS SQL TYPE IS CLOB(n)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 39 on page 510 for more information.
GRAPHIC(n)	fixed-length graphic string	*n* is a positive integer. The maximum value of *n* is 127. See Table 39 on page 510 for more information.
VARGRAPHIC(n)	varying-length graphic string	*n* is a positive integer. The maximum value of *n* is 16 336. See Table 39 on page 510 for more information.

135. In DB2 UDB for UWO, COMP-5 must be used instead of COMP-4.

136. In DB2 UDB for UWO, DISPLAY SIGN LEADING SEPARATE is not supported.

137. Do not use this data type as a column type.

Table 69. SQL Data Types Mapped to Typical COBOL Declarations (continued)

SQL Data Type	COBOL Data Type	Notes
DBCLOB(m)	USAGE IS SQL TYPE IS DBCLOB(m)	*n* is a positive integer. The maximum value of *n* is 1 073 741 823. See Table 39 on page 510 for more information.
BLOB(n)	USAGE IS SQL TYPE IS BLOB(n)	*n* is a positive integer. The maximum value of *n* is 2 147 483 647.
DATE	fixed-length character string	Allow at least 10 characters.
TIME	fixed-length character string	Allow at least 6 characters; 8 to include seconds.
TIMESTAMP	fixed-length character string	Allow at least 19 characters; 26 to include microseconds at full precision.

Notes on COBOL Variable Declaration and Usage

Any level 77 data description entry can be followed by one or more REDEFINES entries. However, the names in these entries cannot be used in SQL statements.

The COBOL declarations for SMALLINT and INTEGER data types are expressed as a number of decimal digits. The database manager uses the full size of the integers and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration. However, this can cause data truncation or size errors when COBOL statements are being executed. The size of numbers in the application must be within the declared number of digits.

Appendix J. Coding SQL Statements in Java Applications

Support for embedded static SQL in Java applications is commonly referred to as "SQLJ". This appendix also makes use of that term.

Defining the SQL Communications Area in Java

A Java program containing SQL statements does not use any Java class corresponding to an SQLCA to inform an application of errors and warnings resulting from the execution of its contained SQL statements. Instead, Java programs are made aware of errors and warnings as described in "Handling SQL Errors and Warnings in Java" on page 639.

Defining SQL Descriptor Areas in Java

A Java program containing SQL statements does not use any Java class corresponding to an SQLDA to associate the application's variables with the input and output parameters of its contained SQL statements. Due to the absense of an SQLDA, none of the SQL statements that include a USING DESCRIPTOR clause are able to specify that clause. Instead, Java programs directly embed host variables and expressions in the SQL statements as described in "Using Host Variables and Expressions in Java" on page 631

Embedding SQL Statements in Java

In a Java program, static SQL statements used for database access are contained in *SQLJ clauses*. SQLJ clauses containing SQL statements are called *executable clauses*. SQLJ clauses that result in declarations of Java classes needed by the executable clauses are called *declaration clauses*, and the classes that result are called *generated classes*.

An executable clause may appear anywhere in a program that a Java statement is permitted. An executable clause begins with the characters #sql, terminates with a semicolon (;), and contains an SQL statement enclosed in braces, {}. The SQL statement itself has no terminating character. An example executable clause is:

```
#sql {DELETE FROM EMPLOYEE};
```

A declaration clause may appear anywhere in a program that a Java class declaration is permitted. An declaration clause begins with the characters #sql, terminates with a semicolon (;), and contains information used in the generation of either an SQLJ database connection context class or an SQLJ iterator class. An example declaration clause is:

```
#sql public iterator DeptSummary (String, String, BigDecimal);
```

This clause results in the generation of a declaration of a public SQLJ iterator class named DeptSummary that fulfills part of the role a cursor declaration does in other application languages. In this example, an associated cursor would be one involving two character strings and a decimal value in that order.

Before any embedded SQL statements can be executed in an application program, code must be included to accomplish these tasks:

© Copyright IBM Corp. 1982, 2002

- Import the Java packages for SQLJ run-time support and the JDBC interfaces used by SQLJ.[138]
- Load the platform-specific JDBC driver.
- Connect to a data source by creating a connection context.
- Optionally, create an execution context.

To import the Java packages for SQLJ and JDBC, these lines are included in the application program:

```
import sqlj.runtime.*;         // SQLJ runtime support
import java.sql.*;             // JDBC interfaces
```

To load the JDBC driver and register it with the `java.sql.DriverManager`, invoke method `Class.forName` with a `java.lang.String` argument identifying the platform's JDBC driver class:
- DB2 UDB for z/OS and OS/390: "COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver".
- DB2 UDB for UWO: "COM.ibm.db2.jdbc.app.DB2Driver" or "COM.ibm.db2.jdbc.net.DB2Driver" ".app." is for applications and ".net." for applets.
- DB2 UDB for iSeries: "com.ibm.db2.jdbc.app.DB2Driver".

For example:

```
try
{
  Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
}
catch (ClassNotFoundException e)
{
  e.printStackTrace();
}
```

A *connection context* specifies the data source each executable clause is to be executed against. This allows an application to direct individual SQL statements to distinct data sources. Connections contexts are described further in “Connecting To, and Using a Data Source” on page 629.

An *execution context* provides access to an executable clause's warning information and in the case of a CALL statement to a procedure's returned result sets. It also allows some attributes of a statement's execution to be controlled, such the
G maximum number or rows returned. The support provided for an execution
G context to control a statement's execution is platform specific. Further details regarding an execution context's use in returning warning information is provided in “Handling SQL Errors and Warnings in Java” on page 639.

In executable clauses, either or both connection contexts and execution contexts are explicitly specified by enclosing them in square brackets, [ ], following the #sql at the beginning of the embedded SQL statement. If both are specified, the connection context is listed first, followed by a comma, followed by the execution context.

Comments

To include comments in an SQLJ program, use either Java comments or SQL comments.

138. SQLJ was designed to coexist with (and in many respects depend on) JDBC. A single application could create a JDBC connection and use that connection to execute dynamic SQL statements through JDBC and embedded static SQL through SQLJ.

- Java comments are denoted by /* */ or //. Java comments can be used outside of SQLJ clauses, wherever the Java language permits them. Within an executable clause, Java comments can only be used in embedded host expressions.
- SQL comments (--) can be used in executable clauses, anywhere except in embedded host expressions.

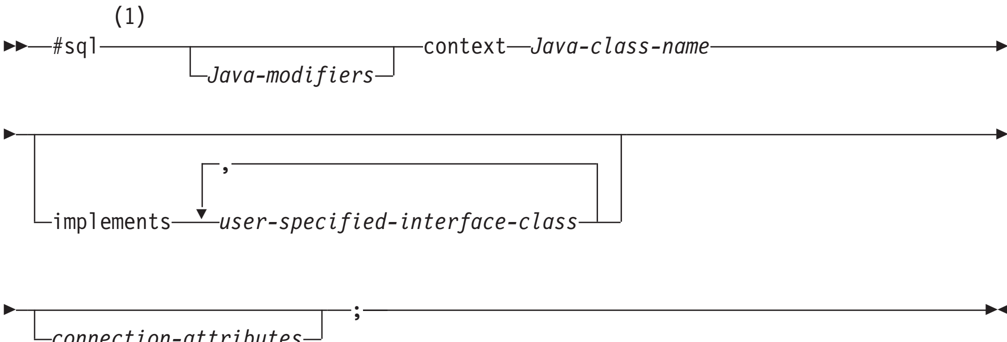
Connecting To, and Using a Data Source

In an SQLJ application, a connection to a data source must be established *before* SQL statements can be executed. A connection to a data source is referred to as a connection context, each of which is an instance of a generated *connection context class* and is declared with a *connection declaration clause*.

Declaring a Connection Context

A connection declaration clause may appear anywhere in a program that a Java class declaration is permitted.

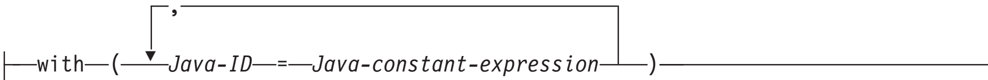
Syntax



Notes:

1 The Java programming language is case-sensitive and lower case is typically used for keywords. For that reason, unlike other SQL keywords, the keywords appearing in a connection context declaration clause are shown in lower case and must appear in the statement in lower case.

connection-attributes:



Description

Java-modifiers
: Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
: Names the generated connection context. *Java-class-name* must be a valid Java identifier.

implements
The clause specifies one or more user-defined Java interfaces that this connection context implements. Each contained *user-specified-interface-class* must identify a valid Java interface according to Java's rules for use of interfaces.

with
Introduces a set of static attributes of the generated connection context class and the initial value of each such static attribute.

Java-ID
Names a user-defined static attribute of a generated connection context class. *Java-ID* must be a valid Java identifier. The value of *Java-constant-expression*, which supplies that attribute's initial value, is also user-defined.

Initiating and Using a Connection

After a connection declaration clause has resulted in the generation of a connection context class and the appropriate JDBC driver has been registered with the DriverManager, to initiate a connection to a data source one of the following methods is used:

- Connection method 1:
  1. Invoke the constructor for the connection context class with the following arguments:[139]
     - a `java.lang.String` that specifies the location name associated with the data source,
     - a boolean that specifies whether autoCommit is on or off for the connection.

  For example, with DB2 UDB for z/OS and OS/390, to use the first method to set up connection context myConn to access data associated with location NEWYORK and to set autoCommit off, the following steps are taken. First, specify a connection declaration clause to generate a connection context class:

  ```
  #sql context Ctx;
  ```

  Then register a JDBC driver and invoke the constructor for generated class Ctx with arguments jdbc:db2os390sqlj:NEWYORK and `false`:

  ```
  Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
  Ctx myConn=new Ctx("jdbc:db2os390sqlj:NEWYORK",false);
  ```

- Connection method 2:
  1. Invoke the JDBC `java.sql.DriverManager.getConnection` method.[140] One form of `java.sql.DriverManager.getConnection` takes a single `java.lang.String` that specifies the location name associated with the data source. The invocation returns an instance of class `java.sql.Connection`, which represents a JDBC connection to that data source.
  2. For environments other than the CICS environment the default state of autoCommit for a JDBC connection is on. To disable autoCommit, invoke the setAutoCommit method on the Connection object with an argument of `false`.
  3. Invoke the constructor for the connection context class. For the argument of the constructor, use the JDBC Connection returned from `java.sql.DriverManager.getConnection`.

---

139. A connection context class has several different constructors. The following describes using only one of them. For further information see applicable product documentation.

140. DriverManager.getConnection has several different signatures. The following describes using only one of them. For further information see applicable product documentation.

For example, with DB2 UDB for z/OS and OS/390, to use the second method to set up connection context myConn to access the data source associated with location NEWYORK with autoCommit off, execute a connection declaration clause to generate a connection context class:

```
#sql context Ctx;
```

Then register a JDBC driver, and invoke `java.sql.Driver.getConnection` with the argument jdbc:db2os390sqlj:NEWYORK, set autoCommit off for the connection, and invoke the constructor for class Ctx using the JDBC connection as the argument:

```
Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
Connection jdbcConn=DriverManager.getConnection("jdbc:db2os390sqlj:NEWYORK");
jdbcConn.setAutoCommit(false);
Ctx myConn=new Ctx(jdbcConn);
```

Connection method 2 results in SQLJ and JDBC sharing the same connection, and is one that may be taken by an application needing both static and dynamic access to the same data source.

Once a connection context is established, to perform an SQL statement at a data source use one of the following two methods:

- Use an *explicit connection.*

  Specify a connection context, enclosed in square brackets, following the #sql. For example, the following executes an UPDATE statement at the data source associated with connection context myConn:

  ```
  #sql [myConn] {UPDATE DEPARTMENT
         SET MGRNO=:hvmgr WHERE DEPTNO=:hvdeptno};
  ```

- Use a *default connection.*

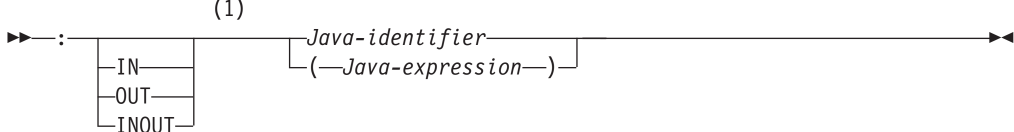
  When an executable clause does not specify a connection context, a default connection context is used. SQLJ's default connection context is implemented by the class sqlj.runtime.ref.DefaultContext. An application's default connection can be set to a specified data source using the sqlj.runtime.ref.DefaultContext.setDefaultContext method, after which that connection will be used as if it had been explicitly specified by any executable clause that does not specify a connection context. Alternatively, if setDefaultContext is not used to override it the default connection will be to the default relational database.

Use of a default connection context is not recommended. The reasons are as follows. First, the default context is not fully specified by the applicable standards, for example its class name sqlj.runtime.ref.DefaultContext is not defined by standard, and any reference to it could result in non-portable applications. Second, the setDefaultContext method is implemented using a static variable which may cause difficulties for reentrant or multi-threaded applications. Use of explicit connections is considered safer.

Using Host Variables and Expressions in Java

Use of a host variable or an expression in embedded SQL is similar to using those variables or expressions in any Java statement, and all of the rules for a Java variable being in scope and declared before it is used apply. There is no requirement that a host variable appear in a declare section and SQLJ supports neither the BEGIN DECLARE SECTION nor the END DECLARE SECTION statements.

Syntax



Notes:

1 The Java programming language is case-sensitive and lower case is typically used for keywords. However, keywords used in embedding a host variable or expression, and outside the expression's embedded *Java-expression*, are considered SQL keywords. These keywords are shown in upper case and able to appear in the statement in any mix of upper or lower case.

In an executable clause a simple variable can be referenced by preceding it with a colon (:). A Java expression can be used by enclosing it in parentheses, '()', and preceding the left parenthesis with a colon. For example to update the yearly bonus of the employee identified by the host variable empID, based on an expression involving the host variable yearsEmployed, one might use:

```
#sql {UPDATE EMPLOYEE
      SET BONUS=:(((int) yearsEmployed/5)*500) WHERE EMPNO=:empID};
```

The expression '((int) yearsEmployed/5)*500' is evaluated with Java's rules for rounding and truncation, and including any side effects that would occur had it appeared outside of an executable clause (for example, had it been '((int) yearsEmployed++/5)*500', yearsEmployed would have been incremented following its use), and the expression's result is the value assigned to BONUS. Note that use of an array is treated as use of an expression, and must be enclosed in parentheses. In other words, if 'hArray' is a Java array object then ':hArray[5]' is not properly formed and must instead be specified as ':(hArray[5])'

When invoking a procedure, it may be necessary to indicate whether a host variable or expression represents an IN, OUT, or INOUT parameter, i.e., to specify a parameter's *parameter mode*. This is done by following the introductory colon with the appropriate IN, OUT, or INOUT keyword. If not specified, the parameter mode is assumed to be IN. Parameter modes must be correct for each parameter of the procedure invoked or the necessary code will not be generated to, for example, assign the value of an OUT parameter to its target host variable. Outside a CALL statement, parameter mode has little meaning. If specified for an input value, then IN may be specified. If parameter mode is specified in a situation where output is involved, for example the INTO portion of a FETCH statement, then OUT may be specified.

Using SQLJ Iterators to Retrieve Rows From a Result Table

The SQLJ equivalent of a cursor is an *SQLJ iterator*. An SQLJ iterator is defined using an *iterator declaration clause*. An SQLJ iterator is either a *positioned iterator* or a *named iterator*. All iterator declaration clauses specify:

- information for its generated Java class declaration, such as whether the iterator is public[141] or static

141. Iterators must be public when an *iterator-attributes* clause is specified.

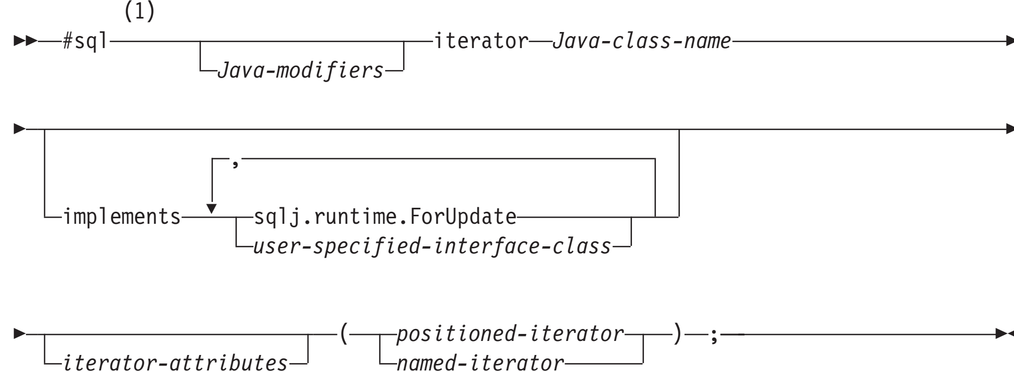
- a set of static attributes, in an *iterator-attributes* clause, such as whether the iterator is holdable or whether columns of its underlying table or view can be updated
- a list of Java data types, and in the case of a named iterator the names of the accessor methods to be used to access the columns of the underlying cursor.

As explained in the next sections, whether the named or positioned type of SQLJ iterator is chosen impacts both how an iterator is declared and how an iterator is used.

Declaring Iterators

An iterator declaration clause may appear anywhere in a program that a Java class declaration is permitted.

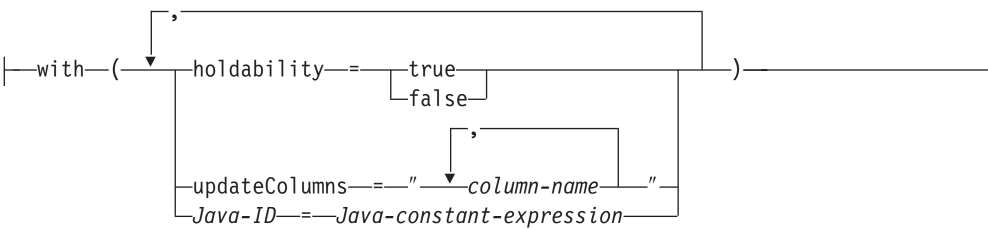
Syntax



Notes:

1 The Java programming language is case-sensitive and lower case is typically used for keywords. For that reason, unlike other SQL keywords, the keywords appearing in an iterator declaration clause are shown in lower case and must appear in the statement in lower case.

iterator-attributes:



positioned-iterator:

named-iterator:

Description

Java-modifiers
Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Names the generated iterator class. *Java-class-name* must be a valid Java identifier.

implements
The implements clause specifies one or more user-defined Java interfaces, or the SQLJ interface sqlj.runtime.ForUpdate, that this iterator supports.

Each contained *user-specified-interface-class* must identify a valid Java interface according to Java's rules for use of interfaces. The iterator must be declared to implement at least the SQLJ interface sqlj.runtime.ForUpdate if it is to be referenced in a positioned UPDATE or positioned DELETE operation.

with
Introduces a set of static attributes of the generated iterator class and the initial value of each such static attribute.

holdability
Specifies a Java boolean value that indicates whether an iterator keeps its position in a table after a COMMIT statement is executed.

updateColumns
Lists the *column-name*s of the underlying table or view allowed to be modified when the iterator is used in a positioned UPDATE statement. The value for updateColumns is a Java String literal containing column names, separated by commas.

If updateColumns is specified in a with element of an iterator declaration clause, the iterator declaration clause must contain an implements clause that includes sqlj.runtime.ForUpdate.

Java-ID
Names a user-defined static attribute of a generated iterator class. *Java-ID* must be a valid Java identifier. The value of *Java-constant-expression*, which supplies that attribute's initial value, is also user-defined.

positioned-iterator
Specifies a list of one or more Java data types. These data types describe the columns of the result table, in left-to-right order.

Java-data-type
The Java data type of a column of the result table of a positioned iterator.

named-iterator
Specifies a list of one or more Java data types and Java accessor method identifiers.

Java-data-type
The Java data type of a column of the result table of the named iterator, and the result data type of the accessor method for that column.

column-accessor
: Names an accessor method for a column of the result table of the named iterator class. *column-accessor* must be a valid Java identifier.

Using Positioned Iterators to Retrieve Rows From a Result Table

A positioned iterator is the type most like a cursor in non-Java applications. The columns of a positioned iterator correspond to the columns of the result table, in left-to-right order. If an iterator declaration clause contains two or more data type declarations, the first corresponds to the first column in the result table, the second to the second column in the result table, and so on.

When an iterator declaration clause for a positioned iterator is encountered, it is replaced with a generated *positioned iterator class* with the name specified in the iterator declaration clause. An object of the positioned iterator's class can then be used to fetch rows from a result table.

For example, suppose rows are to be retrieved from a result table containing the values of the LASTNAME and HIREDATE columns of the table EMPLOYEE. A positioned iterator class is first declared with two columns of the appropriate data types, see "Determining Equivalent SQL and Java Data Types" on page 640 for additional information. The following declares the class ByPos, whose first column is of class `String`, and second of JDBC-defined class `java.sql.Date`. It then declares positer to be object of the ByPos class:

```
#sql iterator ByPos(String,java.sql.Date);
ByPos positer;
```

To use an iterator, an *assignment clause* assigns the result table from a SELECT statement to an instance of an iterator class. Figure 13 shows how positer can be used to retrieve the result table rows.

```
    String name = null;
    Date hrdate;
1   #sql positer = {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
2   #sql {FETCH :positer INTO :name, :hrdate};
                                 // Retrieve the first row
3   while ( !positer.endFetch() )
    {
      System.out.println(name + " was hired on " + hrdate);
      // Retrieve the rest of the rows
      #sql {FETCH :positer INTO :name, :hrdate};
    }
4   positer.close();
```

Figure 13. Retrieving rows using a positioned iterator

Notes to Figure 13:

1 This executable clause performs the SELECT statement, constructs an iterator object containing the result table for the SELECT, and assigns the iterator object to variable positer. In the terminology of other language embeddings this statement performs the functions of both the DECLARE CURSOR and the OPEN statements.

2 The FETCH statement uses left-to-right positional mapping to assign columns of positer's result table to the corresponding variables in the INTO list.

Note that unlike other executable clauses the FETCH statement never needs the iterator's data source to be identified with an explicit connection context. Each instance of an iterator remembers its associated data source.

3 Method endFetch(), a method of the generated iterator class ByPos, returns a value of `true` when all rows have been retrieved from the iterator, and `false` otherwise. The first FETCH statement needs to be executed before endFetch() is called.

4 Method close(), a method of the generated iterator class ByPos, should be called to release resources associated with the iterator when that iterator is no longer needed.

Using Named Iterators to Retrieve Rows From a Result Table

Using named iterators is an alternative way to select rows from a result table. When a named iterator is declared, names are specified that match those of a result table's columns.

When an iterator declaration clause for a named iterator is encountered, it is replaced with a generated *named iterator class* with the name specified in the iterator declaration clause. That generated class includes an accessor method for each column in the iterator declaration clause. The accessor method's name is the name of the column specified in the iterator declaration clause, and its result data type is the data type of the associated column in that clause. As with all Java identifiers an accessor method's name is case sensitive. However, while the accessor method's name is case sensitive, an accessor method's name and a result table column name that differ only in case are considered to be matching names.

The following iterator declaration clause generates the named iterator class ByName, which includes two accessor methods. Those accessor methods are LastName() returning values of class `java.lang.String`, and HireDate() returning values of class `java.sql.Date`. Then nameiter is declared to be an object of the ByName class:

```
#sql iterator ByName(String LastName, java.sql.Date HireDate);
ByName nameiter;
```

To use an iterator, an assignment clause assigns the result table from a SELECT statement to an instance of an iterator class. Figure 14 shows how nameiter could be used to retrieve rows from a result table containing values of the LASTNAME and HIREDATE columns of the EMPLOYEE table.

```
1  #sql nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
2  while (nameiter.next())
   {
     System.out.println( nameiter.LastName() +
                         " was hired on " + nameiter.HireDate());
   }
3  nameiter.close();
```

Figure 14. Retrieving rows using a named iterator

Notes to Figure 14 on page 636:

1 This executable clause performs the SELECT statement, constructs an iterator object containing the result table for the SELECT, and assigns the iterator object to variable nameiter. In the terminology of other language embeddings this statement performs the functions of both the DECLARE CURSOR and the OPEN statements.

2 Method next(), a method of the generated class ByName, replaces the FETCH statement of positioned iterators. It advances the iterator to successive rows of the result set. `next` returns a value of `true` when a next row is available, and a value of `false` when all rows have been fetched.

3 Method close(), a method of the generated iterator class ByName, should be called to release resources associated with the iterator when that iterator is no longer needed.

The names of a named iterator's accessor methods must be valid Java identifiers. The names must also match the column names in the result table from which the iterator retrieves its rows. If a SELECT statement that will be assigned to a named iterator involves columns that either have no names or whose names might not be valid Java identifiers, the SQL AS clauses can be used to give columns of the result table acceptable names.

For example, suppose a named iterator is to be used to retrieve the rows specified by this statement:

```
SELECT PUBLIC FROM GOODTABLE
```

The iterator column name must match the column name of the result table, but a name of `public` cannot be specified because `public` is a reserved Java keyword. This leaves one of two choices. First, because Java is case sensitive, the iterator could declare that a name such as puBlic, or PUBlic, or PUBLIC be given to the `PUBLIC` column, or an AS clause could be used to rename `PUBLIC` to a Java identifier that is not similar to a keyword. For example:

```
SELECT PUBLIC AS IS_PUBLIC FROM GOODTABLE
```

A named iterator with a column name that is a valid Java identifier and matches the column name of the result table can then be declared:

```
#sql iterator ByName(String is_public);
ByName nameiter;
```

And nameiter could then be used as the target of an assignment clause:

```
#sql nameiter={SELECT PUBLIC AS IS_PUBLIC FROM GOODTABLE};
```

Using Iterators For Positioned UPDATE and DELETE Operations

When declaring an iterator that will be used in a positioned UPDATE or DELETE statement, an SQLJ *implements clause* is used to specify that the iterator implements the `sqlj.runtime.ForUpdate` interface. The iterator must also be declared as public. For example, suppose instances of iterator class ByPos are to be used in a positioned DELETE statement. The declaration would be:

```
#sql public iterator ByPos(String) implements sqlj.runtime.ForUpdate
  with(updateColumns="EMPNO");
```

Because the iterator is public but not static Java requires that it either be declared in a different source file, or be declared as a nested class. To use the iterator when it is declared in a different source file:

1. Import the generated iterator class.

2. Declare an instance of the generated iterator class.
3. Assign the SELECT statement associated with the positioned UPDATE or DELETE to the iterator instance.
4. Execute positioned UPDATE or DELETE statements using the iterator.

After the iterator is created, any application that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator. The authorization ID under which a positioned UPDATE or DELETE statement executes is the authorization ID under which the DB2 package containing the UPDATE or DELETE executes.

For example, consider the named iterator UpdByName declared in the following example.

```
#sql public iterator UpdByName(String EMPNO, BigDecimal SALARY)
  implements sqlj.runtime.ForUpdate
  with(updateColumns="SALARY");
```

To use UpdByName for a positioned UPDATE in another file, execute statements like those in Figure 15.

```
1   import UpdByName;
        :
        :
    {
      UpdByName upditer;         // Declare object of UpdByName class
      String enum;
2     #sql upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
                        WHERE WORKDEPT='D11'};
3     while (upditer.next())
      {
        enum = upditer.EMPNO();  // Get value from result table
4       #sql {UPDATE EMPLOYEE SET SALARY=SALARY*1.05 WHERE CURRENT OF :upditer};
                                 // Update row where cursor is positioned
        System.out.println("Updating row for " + enum);
      }
      upditer.close();           // Close the iterator
      #sql {COMMIT};             // Commit the changes
    }
```

Figure 15. Updating rows using a positioned iterator

Notes to Figure 15:

1 This statement imports named iterator class UpdByName, generated by the iterator declaration clause for UpdByName in UpdByName.sqlj. The import command is not needed if UpdByName is in the same package as the Java source file that references it.

2 This executable clause performs the SELECT statement, constructs an iterator object containing the result table for the SELECT, and assigns the iterator object to variable upditer.

3 This statement positions the iterator to the next row to be updated.

4 This executable clause performs the positioned UPDATE.

Handling SQL Errors and Warnings in Java

A Java program containing SQL statements does not use an SQLCA or support the WHENEVER statement. SQLJ throws an Exception of the JDBC-defined class java.sql.SQLException whenever an SQL statement returns an error. To handle SQL errors, import java.sql.SQLException and use the Java language try/catch blocks to modify program flow when an SQL error is returned. After an exception is caught, the SQLException's getErrorCode method can be used to retrieve a return code and its getSQLState method to retrieve SQLSTATE values. For example, the following SELECT INTO statement would fail and an SQLException would be thrown if more than one row exists for the employee whose EMPNO is '000010':

```
try
{
  #sql {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
}
catch(SQLException e)
{
  System.out.println("SQLSTATE returned: " + e.getSQLState());
}
```

Unlike errors, warnings returned by SQL do not result in SQLExceptions. The handling of warnings depends on whether the warning is associated with an executable clause or with an SQLJ iterator. In either case, first import java.sql.SQLWarning.

To check for a warning associated with an executable clause, after the clause is executed invoke the getWarnings method against the execution context associated with that clause. getWarnings returns the first warning an SQL statement generates. Subsequent warnings are chained to the first SQLWarning. An execution context can either be explicitly specified in the embedded SQL statement or accessed from the connection context associated with the statement. The following example retrieves an SQLWarning, with execution context ExecCtx specified explicitly:

```
ExecutionContext ExecCtx = new ExecutionContext();
#sql [ExecCtx] {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
SQLWarning sqlWarn = ExecCtx.getWarnings();
if (sqlWarn != null)
   System.out.println("SQLWarning " + sqlWarn);
```

Alternatively, to access the execution context associated with connection context myConn:

```
#sql [myConn]  {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
ExecutionContext ExecCtx = myConn.getExecutionContext();
SQLWarning sqlWarn = ExecCtx.getWarnings();
if (sqlWarn != null)
   System.out.println("SQLWarning " + sqlWarn);
```

To check for a warning associated with an SQLJ iterator, invoke the generated iterator class's getWarnings method against the iterator. To be aware of all warnings, it is necessary for the getWarnings method to be invoked following each fetch operation. The overhead of those invocations should be weighed against the possible benefit of knowing a warning has been reported. It may be useful to test for warnings only if there is corrective action that an application will take following a warning. In that case, then if, for example, an SQLJ iterator has been declared:

```
#sql positer = {SELECT LASTNAME, SALARY FROM EMPLOYEE};
```

Then an application could test for warnings as shown in the following:

```
#sql {FETCH :positer INTO :name, :sal};
while ( !positer.endFetch() )
{
  SQLWarning sqlWarn = positer.getWarnings();
  if (sqlWarn != null)
     System.out.println("SQLWarning " + sqlWarn);
  System.out.println( name + " has base salary " + sal );
  #sql {FETCH :positer INTO :name, :sal};
}
positer.close();
```

Note that the end of data condition for a result set does not cause getWarnings to report a warning.

An important subclass of both `java.sql.SQLException` and `java.sql.SQLWarning` is that of `java.sql.DataTruncation`. A `java.sql.DataTruncation` exception may be thrown when an update operation storing or modifying data causes a data truncation error to be returned. Alternatively, a `java.sql.DataTruncation` may be reported through `getWarnings()` when a truncation takes place reading data from a data source.

The `java.sql.DataTruncation` class supports methods providing information specific to truncation errors or warnings that is not otherwise available through `java.sql.SQLException` and `java.sql.SQLWarning`. For further information see applicable product documentation.

Determining Equivalent SQL and Java Data Types

Table 70. Equivalent Java and SQL data types

Java data type	SQL data type
short, java.lang.Integer	SMALLINT
int, java.lang.Integer	INTEGER
java.math.BigDecimal	DECIMAL, NUMERIC
float, java.lang.Float	REAL
double, java.lang.Double	DOUBLE
java.lang.String	CHAR, VARCHAR, GRAPHIC, VARGRAPHIC
byte[][1]	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA
java.sql.Blob[2]	BLOB
java.sql.Clob[2]	CLOB
java.sql.Date[2]	DATE
java.sql.Time[2]	TIME
java.sql.Timestamp[2]	TIMESTAMP

Note:

1. Because this data type is equivalent to a DB2 character data type defined as FOR BIT DATA, SQLJ performs no character conversion for data of this type.
2. This class is part of the JDBC API.

Example

The following example, using DB2 UDB for z/OS and OS/390, solicits the name of a department, obtains the names and phone numbers of all members of that department from the EMPLOYEE table, and presents that information on the screen.

```
package Reports;

import sqlj.runtime.*;
import java.sql.*;
import java.io.*;
import COM.ibm.db2os390.sqlj.jdbc.*;

#sql context CT1x;

public class Summary
{
  #sql static iterator ReportDept(String lastName, String phoneNo);

  /* Names and Phones by Department */
  public static void main (String[] args) // Main entry point
  throws SQLException
  {
    CT1x myConn=null;
    InputStreamReader inStream = new InputStreamReader(System.in);
    char[] inBuffer = new char[10];
    int ii;
    String workDept;
    ReportDept deptSummary = null;  /* iterator used to process the select */

    /* Get a local connection from DB2 for OS/390 JDBC driver, with */
    /* autocommit off. For any errors in setup, print a stack trace and exit. */
    try
    {
      Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
      myConn=new CT1x("jdbc:db2os390sqlj:", false);
    }
    catch (SQLException e)
    {
      e.printStackTrace();
      return;
    }
    catch (ClassNotFoundException e)
    {
      e.printStackTrace();
      return;
    }

    try
    {
      /* Get the department number to be used in the SELECT statement and */
      /* put into upper case. */
      System.out.println( "Enter a Department number, followed by a <return> ");
      ii = inStream.read(inBuffer, 0, 10);
      inStream.close();
      workDept = (new String(inBuffer)).trim().toUpperCase();

      /* Perform the select */
      #sql [myConn] deptSummary =
         {SELECT LASTNAME,PHONENO FROM EMPLOYEE WHERE WORKDEPT = :workDept};

      System.out.println("Here are the members of Department " + workDept);
      /* For all rows in the result table */
      while (deptSummary.next())
      {
        /* Display name and phone. If employee does not have a phone, */
```

```
          /* then display ? */
          if (deptSummary.phoneNo() == null)
            System.out.println( deptSummary.lastName() + "  ?");
          else
            System.out.println( deptSummary.lastName() + "  " +
                                deptSummary.phoneNo());
        }

        /* Close the cursor and end the logical unit of work */
        deptSummary.close();
        #sql [myConn] {COMMIT};
      }
      catch (SQLException e)
      {
        e.printStackTrace();
        return;
      }
      catch (java.io.IOException)
      {
        e.printStackTrace();
        return;
      }
      finally
      {
        /* whether an error occurred or not, close any created connection */
        if (myConn != null)
          myConn.close();
      }
    }  /* main */
  }  /* Summary */
```

Appendix K. Coding SQL Statements in REXX Applications

In the HP-UX, Linux, and Solaris environments, REXX is not supported.

SQL is enabled in REXX through the special REXX command EXECSQL, which is used to pass SQL statements to the database manager for processing. [142]

REXX procedures do not have to be preprocessed. At runtime, the REXX interpreter passes SQL statements to the database manager for processing.

The SQL/REXX interface supports the following SQL statements:

ALTER TABLE [143]
CALL [145]
CLOSE
COMMIT
COMMENT [143]
CREATE ALIAS [143]
CREATE DISTINCT TYPE [143]
CREATE FUNCTION [143]
CREATE INDEX [143]
CREATE PROCEDURE [143]
CREATE TABLE [143]
CREATE VIEW [143]
DECLARE CURSOR
DELETE [143, 144]
DESCRIBE
DROP [143]
EXECUTE
EXECUTE IMMEDIATE
FETCH
GRANT [143]
INSERT [143, 144]
LOCK TABLE [143]
OPEN
PREPARE
RENAME [143]
REVOKE [143]
ROLLBACK
SET PATH [143, 144]
UPDATE [143, 144]

The following SQL statements are not supported by the SQL/REXX interface:

BEGIN DECLARE SECTION
CONNECT
END DECLARE SECTION
FREE LOCATOR
INCLUDE
RELEASE
SELECT INTO
SET CONNECTION
SET transition-variable
VALUES
VALUES INTO
WHENEVER [146]

---

142. In DB2 UDB for UWO in the OS/2, AIX and Windows for 32-bit operating systems environments, the database manager supports REXX through calls to an external function named SQLEXEC. In the examples that follow, where EXECSQL '...' appears, substitute CALL SQLEXEC '...' in these environments.

143. In DB2 UDB for UWO in the OS/2, AIX and Windows for 32-bit operating systems environments, this statement is supported via either PREPARE followed by EXECUTE, or by EXECUTE IMMEDIATE.

144. These statements cannot be executed directly if they contain host variables; they must be the object of a PREPARE and then an EXECUTE.

145. The CALL statement cannot include host variables or the USING DESCRIPTOR clause.

146. See “Handling SQL Errors and Warnings in REXX” on page 647 for more information.

© Copyright IBM Corp. 1982, 2002

Defining the SQL Communications Area in REXX

The fields that make up the SQL Communications Area (SQLCA) are automatically included by the SQL/REXX interface. An INCLUDE SQLCA statement is not required, nor is it allowed. The SQLSTATE or SQLCODE fields of the SQLCA contain SQL return codes. These values are set by the database manager after each SQL statement is executed. An application can check the SQLSTATE or SQLCODE value to determine whether the last SQL statement was successful.

The SQL/REXX interface uses the SQLCA in a manner consistent with the typical SQL usage. (See Appendix C, “SQL Communication Area (SQLCA)” on page 525 for more information.) However, the SQL/REXX interface maintains the fields of the SQLCA in separate variables rather than in a contiguous data area. The variables that the SQL/REXX interface maintains for the SQLCA are defined as follows: [147]

SQLCODE
The SQL return code.

SQLERRMC
Error and warning message tokens.

SQLERRP
Product code and, if there is an error, the name of the module that returned the error.

SQLERRD.*n*
Six variables (*n* is a number between 1 and 6) containing diagnostic information.

SQLWARN.*n*
Eleven variables (*n* is a number between 0 and 10) containing warning flags.

SQLSTATE
An SQL return code that indicates the outcome of the most recently executed SQL statement. Portable applications should use the SQLSTATE return code instead of SQLCODE return code.

Defining SQL Descriptor Areas in REXX

The following statements require an SQLDA:

CALL...USING DESCRIPTOR *descriptor-name*

DESCRIBE *statement-name* INTO *descriptor-name*

EXECUTE...USING DESCRIPTOR *descriptor-name*

FETCH...USING DESCRIPTOR *descriptor-name*

OPEN...USING DESCRIPTOR *descriptor-name*

PREPARE *statement-name* INTO *descriptor-name* ...

Unlike the SQLCA, there can be more than one SQLDA in a procedure, and an SQLDA can have any valid name. Each SQLDA consists of a set of REXX variables with a common stem, where the name of the stem is the *descriptor-name* from the appropriate SQL statement(s). This must be a simple stem; that is, the stem itself must not contain any periods. The SQL/REXX interface automatically provides the fields of the SQLDA for each unique descriptor name. An INCLUDE SQLDA statement is not required, nor is it allowed.

The SQL/REXX interface uses the SQLDA in a manner consistent with the typical SQL usage. (See Appendix D, “SQL Descriptor Area (SQLDA)” on page 529 for

147. In DB2 UDB for UWO in the OS/2 environment, the stem SQLCA precedes each SQLCA variable name (such as SQLCA.SQLCODE, SQLCA.SQLERRMC).

more information.) However, the SQL/REXX interface maintains the fields of the SQLDA in separate variables rather than in a contiguous data area.

The following variables are returned to the application after a DESCRIBE statement or a PREPARE statement that contains an INTO clause:

stem.n.SQLNAME
The name of the *n*th column in the result table.

stem.SQLCCSID
The CCSID of the *n*th column of data.

The following variables must be provided by the application before an OPEN...DESCRIPTOR, a FETCH...DESCRIPTOR, or an EXECUTE...DESCRIPTOR statement. They are returned to the application after a DESCRIBE statement or a PREPARE statement that contains an INTO clause:

stem.SQLD
Number of variable elements that the SQLDA actually contains.

stem.n.SQLTYPE
An integer representing the data type of the *n*th element (for example, the first element is in stem.1.SQLTYPE).

The following data types are not allowed:

400/401
NUL-terminated graphic string

404/405
BLOB

408/409
CLOB

412/413
DBCLOB

460/461
NUL-terminated character string

504/505
DISPLAY SIGN LEADING SEPARATE

960/961
BLOB locator

964/965
CLOB locator

968/969
DBCLOB locator

stem.n.SQLLEN
If SQLTYPE does not indicate a DECIMAL or NUMERIC data type, the maximum length of the data contained in stem.n.SQLDATA.

stem.n.SQLLEN.SQLPRECISION
If the data type is DECIMAL or NUMERIC, this will contain the precision of the number.

stem.n.SQLLEN.SQLSCALE
If the type is DECIMAL or NUMERIC, this will contain the scale of the number.

The following variables must be provided by the application before an EXECUTE...DESCRIPTOR or OPEN...DESCRIPTOR statement, they are returned to

the application after a FETCH...DESCRIPTOR statement. They are not used after a DESCRIBE statement or a PREPARE statement that contains an INTO clause:

stem.n.SQLDATA
This contains the input value supplied by the application, or the output value fetched by SQL.

This value is converted to the attributes specified in SQLTYPE, SQLLEN, SQLPRECISION, and SQLSCALE.

stem.n.SQLIND
If the input or output value is null, this will be a negative number.

Embedding SQL Statements in REXX

An SQL statement can be placed anywhere a REXX command can be placed.

Each SQL statement in a REXX procedure must begin with EXECSQL [148] (in any combination of uppercase and lowercase letters), followed by either:

- The SQL statement enclosed in single or double quotes, or
- A REXX variable containing the statement. Note that a colon must not precede a REXX variable when it contains an SQL statement.

For example:

```
EXECSQL "COMMIT"
```

is equivalent to:

```
rexxvar = "COMMIT"
EXECSQL rexxvar
```

The command follows normal REXX rules. For example, it can optionally be followed by a semicolon to allow a single line to contain more than one REXX statement. REXX also permits command names to be included within single quotes; for example:

```
'EXECSQL COMMIT'
```

G In the OS/2 environment, because CALL SQLEXEC is used in place of the
G EXECSQL command, the preceding example is not applicable.

Comments

Neither SQL comments (--) nor REXX comments are allowed in strings representing SQL statements. Otherwise, normal REXX commenting rules are followed.

Continuation of SQL Statements

The string containing an SQL statement can be split into several strings on several lines, separated by commas or concatenation operators, according to standard REXX usage.

Including Code

Unlike the other host languages, support is not provided for including externally defined statements.

148. In the OS/2 environment, EXECSQL is replaced with CALL SQLEXEC.

Margins

There are no special margin rules for the SQL/REXX interface.

Names

Any valid REXX name not ending in a period (.) can be used for a host variable.

Do not use host variable names that begin with 'SQL', 'DB2', 'RDI', 'DSN', 'RXSQL', or 'QRW'. These names are reserved for the database manager.

In DB2 UDB for z/OS and OS/390 and DB2 UDB for UWO in the OS/2 environment, cursor names and statement names are predefined. These predefined names must be used in SQL statements that reference cursors and prepared statement names, and they must not be used as host variable names. See the product documentation for more information.

Nulls

Although the term *null* is used in both REXX and SQL, the term means different things in the two languages. REXX has a null string (a string of length zero) and a null clause (a clause consisting only of blanks and comments). The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a (nonnull) value.

Statement Labels

REXX command statements can be labeled as usual.

Handling SQL Errors and Warnings in REXX

The WHENEVER statement is not supported by the SQL/REXX interface. Any of the following may be used instead:

- A test of the REXX SQLSTATE or SQLCODE variables after each SQL statement to detect error and warning conditions issued by the database manager, but not for those issued by the SQL/REXX interface.
- A test of the REXX RC variable after each SQL statement to detect error and warning conditions. Each use of the EXECSQL command sets the RC variable to:

  0 Statement completed successfully.

  positive A SQL warning occurred.

  negative An SQL error occurred

  This can be used to detect errors and warnings issued by either the database
G manager or by the SQL/REXX interface. The values of RC are product-specific.

G In DB2 UDB for UWO, the RC variable is not set to a positive value for
G warnings.
- The REXX SIGNAL ON ERROR and SIGNAL ON FAILURE facilities can be used to detect errors, but not warnings. This is driven by the REXX RC variable.
G In DB2 UDB for UWO, SIGNAL ON ERROR and SIGNAL ON FAILURE cannot
G be used to detect SQL errors.

Isolation Level

To use different isolation levels in REXX see the product documentation.

Using Host Variables in REXX

REXX does not provide for variable declarations. New variables are recognized by their appearance in assignment statements. Therefore, there is no SQL declare section, and the BEGIN DECLARE SECTION and END DECLARE SECTION statements are not supported.

All host variables within an SQL statement must be preceded by a colon (:).

The SQL/REXX interface performs substitution in compound variables before passing statements to the database manager. For example:

```
a = 1
b = 2
EXECSQL 'OPEN c1 USING :x.a.b'
```

will cause the contents of x.1.2 to be passed to SQL.

Determining Data Types of Input Host Variables

All data in REXX is in the form of strings. The data type of input host variables (that is, host variables used in a 'USING host variable' clause in an EXECUTE or OPEN statement) is inferred by the database manager at run-time from the contents of the variable according to Table 71.

These rules define either numeric, character, or graphic values. A numeric value can be used as input to a numeric column of any type. A character value can be used as input to a character column of any type, or to a date, time, or timestamp column. A graphic value can be used as input to a graphic column of any type.

Table 71. Determining Data Types of Host Variables in REXX

Host Variable Contents	Assumed Data Type	SQL Type Code	SQL Type Description
A number with neither decimal point nor exponent. It can have a leading plus or minus sign.	signed integers	496/497	INTEGER
A number that includes a decimal point, but no exponent, or A number that does not include a decimal point or an exponent and is greater than 2147483647 or smaller than -2147483647. It can have a leading plus or minus sign. m is the total number of digits in the number. n is the number of digits to the left of the decimal point (if any).	packed decimal	484/485	DECIMAL(m,n)
A number that is in scientific or engineering notation (that is, followed immediately by an 'E' or 'e', an optional plus or minus sign, and a series of digits). It can have a leading plus or minus sign.	floating point	480/481	FLOAT
A string with leading and trailing apostrophes (’) or quotation marks ("), which has length n after removing the two delimiters, or A string of length n which cannot be recognized as numeric or graphic via other rules in this table.	varying-length character string	448/449	VARCHAR(n)

Table 71. Determining Data Types of Host Variables in REXX (continued)

Host Variable Contents	Assumed Data Type	SQL Type Code	SQL Type Description
A string with a leading and trailing apostrophe (') or quotation marks (") preceded by the character 'G', 'g', 'N', or 'n', which contains n DBCS characters. [149]	varying-length character string	464/465	VARGRAPHIC(n)
undefined variable	variable for which a value has not been assigned	none	Data that is not valid was detected.

The Format of Output Host Variables

It is not necessary to determine the data type of an *output host variable* (that is, a host variable used in an 'INTO host variable' clause in a FETCH statement). Output values are assigned to host variables as follows:

- Character values are assigned without leading and trailing apostrophes.
- Graphic values are assigned without a leading G or apostrophe, without a trailing apostrophe, and without shift-out and shift-in characters.
- Numeric values are translated into strings.
- Integer values do not retain any leading zeros. Negative values have a leading minus sign. Positive values do not have a leading plus sign.
- Decimal values retain leading and trailing zeros according to their precision and scale. Negative values have a leading minus sign. Positive values do not have a leading plus sign.
- Floating-point values are in scientific notation, with one digit to the left of the decimal place. The 'E' is in uppercase.

Avoiding REXX Conversion

To guarantee that a string is not converted to a number or assumed to be of graphic type, strings can be enclosed in the following: "'". Simply enclosing the string in apostrophes does not work. For example:

```
stringvar = '100'
```

will cause REXX to set the variable *stringvar* to the string of characters 100 (without the apostrophes). This will be evaluated by the SQL/REXX interface as the number 100, and it will be passed to SQL as such.

On the other hand,

```
stringvar = "'"100"'"
```

will cause REXX to set the variable *stringvar* to the string of characters '100' (with the apostrophes). This will be evaluated by the SQL/REXX interface as the string 100, and it will be passed to SQL as such.

Indicator Variables in REXX

An indicator variable is an integer. On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to indicate that a null value should be assigned.

149. In EBCDIC implementations, the byte immediately following the leading apostrophe or quote is a X'0E' shift-out, and the byte immediately preceding the trailing apostrophe or quote is a X'0F' shift-in.

Unlike other languages, a valid value must be specified in the host variable even if its associated indicator variable contains a negative value.

See "References to Host Variables" on page 85 for more information on using indicator variables.

Example

The following example solicits the name of a department, obtains the names and phone numbers of all members of that department from the EMPLOYEE table, and presents that information on the screen.

```
/* Names and Phones by Department Exec */

/* If there are any nonzero return codes, then branch to the error handler */
     Signal on error

/* Prepare the select statement */
     stmt = 'SELECT LASTNAME, PHONENO FROM EMPLOYEE WHERE WORKDEPT = ?'
     what_stmt = 'PREPARE'
     EXECSQL 'PREPARE stmt_name FROM :stmt'

/* Declare the cursor to be used for reading the result table */
     what_stmt = 'DECLARE'
     EXECSQL 'DECLARE c1 CURSOR FOR stmt_name'

/* Get the department number to be used in the SELECT and put into upper case) */
     Say 'Enter a Department number'
     Parse upper pull dept
/* Find all rows that satisfy the SELECT */
     what_stmt = 'OPEN'
     EXECSQL 'OPEN c1 USING :dept'

/* Turn off the automatic error trap (in order to handle FETCH warnings in-line */
     Signal off error

/* For all rows in the result table */
     Say 'Here are the members of Department' dept
     Do forever
       /* Fetch the row */
            what_stmt = 'FETCH'
            EXECSQL 'FETCH c1 INTO :name, :phone :phone_ind'
       /* If no more rows, then done */
            If rc <> 0 & sqlcode = 100 then
               Leave
       /* If error then go to error handler */
            If rc <> 0 then
               Signal error
       /* If employee does not have a phone, then set phone to ? */
            If phone_ind < 0 then
               phone = '?'
       /* Display name and phone */
            Say name phone
     End

/* Turn on the automatic error trap again */
     Signal on error

/* Close the cursor and end the logical unit of work */
     what_stmt = 'CLOSE'
     EXECSQL 'CLOSE c1'

     what_stmt = 'COMMIT'
     EXECSQL 'COMMIT'

     Exit 0
```

```
Error:  /* Error handler */
     Signal off error
     Say ' '
     Say 'Error accessing EMPLOYEE table'
     Say 'Statement in error was:' what_stmt
     Say 'RC      =' rc
     Say 'SQLCODE =' sqlcode
     Exit rc
```

Appendix L. Coding Programs for use by External Routines

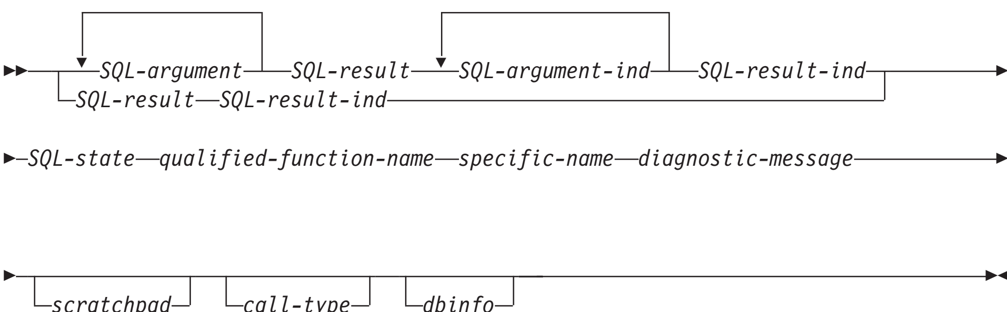
Parameter Passing for External Routines

An external routine invokes an executable program that must be written to accept parameters according to the specified language and parameter style of the routine.

G Whether the program is written as a main program or a subroutine may be
G specified by the PROGRAM TYPE clause (see "CREATE PROCEDURE (External)"
G on page 341) or is product specific for the type of routine.

Parameter Passing for External Functions Written in C or COBOL

An external function written in C or COBOL must use a parameter style of DB2SQL. When using the DB2SQL parameter style, the database manager passes implicit parameters to the program in addition to the parameters specified in the invocation of the user-defined function. The parameters are passed to the program in the order defined by the following diagram.



SQL-argument
: Each *SQL-argument* represents one input parameter defined when the function was created.

  Each input parameter of the function is set by the database manager before invoking the program. The value of each of these arguments is taken from the expression specified in the function invocation. It is assigned to the corresponding parameter definition in the CREATE statement using storage assignment as described in "Assignments and Comparisons" on page 58.

  These arguments are input only and any changes to these argument values made by the program are ignored by the database manager upon return from the program.

SQL-result
: This output argument is the result of the function which must be set by the program before returning to the database manager.

  If the CAST FROM clause was specified in the CREATE FUNCTION statement, the program is expected to return a data type based on the SQL data type specified immediately following the CAST FROM. Then, the database manager does a second CAST, to the SQL data type specified immediately following the

© Copyright IBM Corp. 1982, 2002

RETURNS. If the CAST FROM clause was not specified in the CREATE FUNCTION statement, the program is expected to return a data type based on the SQL data type specified immediately following the RETURNS keyword.

The program must return a value that corresponds to the data type and length of the result as specified when the function was created. See "Attributes of the Arguments Passed to a Routine Program" on page 660 for appropriate data type declarations.

SQL-argument-ind
There is an *SQL-argument-ind* for each *SQL-argument* passed to the program. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* and indicates whether the *SQL-argument* has a value or is NULL.

Each *SQL-argument-ind* is defined as a two-byte signed integer.

Each *SQL-argument-ind* associated with an argument of the function is set by the database manager before invoking the program. It contains one of the following values:

0 The argument is present and not NULL.

-1 The argument value is NULL.

If the function is defined with RETURNS NULL ON NULL INPUT, the program does not need to check for a NULL value. However, if it is defined with CALLED ON NULL INPUT, any argument can be NULL and the program should check each *SQL-argument-ind*.

SQL-result-ind
The output result indicator of the function which must be set by the program before returning to the database manager. The result indicator is used by the program to indicate if the result value is NULL:

0 or positive
The result value is present and not NULL.

negative
The result value is NULL.

Any negative value for the indicator set by the program is returned by the database manager as a -1, except for a value of -2 which is returned as a -2.

SQL-result-ind is defined as a two-byte signed integer.

SQL-state
This output argument is a CHAR(5) value that represents the SQLSTATE. This argument is passed in from the database manager with the initial value set to '00000' and can be set by the program as the result SQLSTATE for the function. While normally the SQLSTATE is not set by the program, it can be used to return an error or warning as follows:

01Hxx
The program detected a warning situation. This results in an SQL warning. Here *xx* may be one of several possible strings.

38xxx
The program detected an error situation. It results in an SQL error. Here *xxx* may be one of several possible strings.

See Appendix E, "SQLSTATE Values—Common Return Codes" on page 539 for more information about valid SQLSTATEs that the program may use.

qualified-function-name
This input argument is set by the database manager before invoking the program. It is a VARCHAR(517) value that contains the name of the function that is invoking the program. The format of the value in *qualified-function-name* is:

```
"schema-name"."function-name"
```

Note that any double quote character within the *schema-name* or *function-name* gets doubled. This argument is useful when the program is being used by multiple function definitions so that the program can distinguish which function is being invoked. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

specific-name
This input argument is set by the database manager before invoking the program. It is a VARCHAR(128) value that contains the specific name of the function that is invoking the program. Like *qualified-routine-name*, this parameter is useful when the program is being used by multiple function definitions so that the program can distinguish which definition is being invoked. See “CREATE FUNCTION” on page 310 for more information about *specific-name*. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

diagnostic-message
This output argument is a VARCHAR(70) value that can be used by the program to send message text back when an SQLSTATE warning or error is returned by the program. It is initialized by the database manager to an empty string before invoking the program and may be set by the program with descriptive information. The *diagnostic-message* argument value is ignored by the database manager unless the *SQL-state* argument is set by the program to a value other than ’00000’.

scratchpad
This input and output argument is set by the database manager before invoking the program. It is only present if the CREATE FUNCTION statement specified the SCRATCHPAD keyword. The scratchpad provides the program access to storage that is persistent across function invocations within the same SQL statement.

This argument is a structure with the following elements:

- an INTEGER containing the length of the scratchpad
- the actual scratchpad, initialized to all binary zeroes by the database manager before the first invocation of the program.

The value of the scratchpad is unchanged by the database manager between invocations of program based on iterations of the same function invocation within an SQL statement.

call-type
This input argument is set by the database manager before invoking the program. It is only present if the CREATE FUNCTION statement for the function specified the FINAL CALL keyword. It is an INTEGER value that contains one of the following values:

-1 This is the first invocation of the program for this statement. A first call is a normal call in that all the external function argument values of the parameter style are passed.

0 This is a normal call. All the external function argument values of the parameter style are passed.

1 This is a final call. No *SQL-argument* or *SQL-argument-ind* values are passed. The program should not set the *SQL-result* or *SQL-result-ind* arguments for a final call since both of these are ignored by the database manager upon return from the program. However, the program may set the *SQL-state* and *diagnostic-message* arguments. These arguments are handled in a way similar to other invocations of the function. The *call-type* argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

dbinfo

This input argument is set by the database manager before invoking the program. It is only present if the CREATE FUNCTION statement for the routine specifies the DBINFO keyword. The argument is a structure whose definition is described in "Database Information in External Routines (DBINFO)" on page 662. The *dbinfo* argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

Parameter Passing for External Functions Written in Java

The Java parameter style is the style specified by *Information technology - Database languages - SQL - Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)* ISO/IEC 9075-13:2002. When coding a Java method for an external function, the following conventions must be followed.

- The Java method must be a public static method.
- The parameters of the Java method must be a Java type that is equivalent to the SQL data type of the parameter (see Table 72 on page 661).
- The Java method must return a Java type that is equivalent to the SQL data type of the result defined for the function (see Table 72 on page 661). The return value is the result of the method.

Consider an example of a function created with parameters of SQL types t1, t2, and t3 and returning type t4 with external name 'jarfile.fname' (jarfile is the Java class name). The database manager will invoke the Java method with the expected Java signature:

```
public static T4 fname (T1 a, T2 b, T3 c) { .....}
```

Where:

- fname is the Java method name
- T1 through T4 are the Java types that correspond to SQL types t1 through t4.
- a, b, and c are arbitrary variable names for the input arguments.

For example, given an external function called sample.test3 that returns INTEGER and takes arguments of type CHAR(5), INTEGER, and DATE, the database manager expects the Java implementation of the function to have the following signature:

```
import java.sql.*;
public class sample
{
  public static int test3(String arg1, int arg2, Date arg3) { ... }
}
```

To return a result of an external function from a Java method when using the JAVA parameter style, simply return the result from the method.

```
{
  ...
  return value;
}
```

SQL NULL values are represented by uninitialized Java variables that are non-primitive types. The following primitive Java types do not support the SQL NULL value: short, int, long, float, double. If a null value is passed to a parameter that is a Java primitive type, an SQL error is returned.

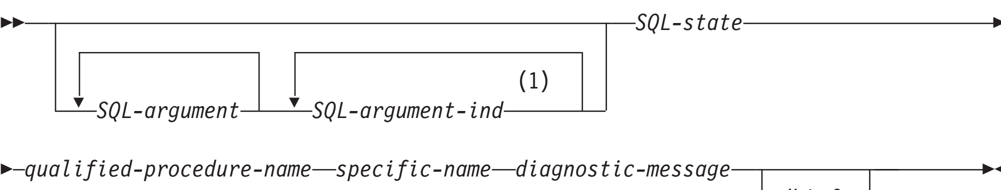
Parameter Passing for External Procedures Written in C or COBOL

An external procedure written in C or COBOL can be defined to use one of three parameter styles. When using the DB2SQL parameter style, the database manager passes parameters to the program in addition to the parameters specified in the call to the procedure. Depending on the parameter style, the parameters are passed to the program in the order defined by the following diagrams.

Parameter Style GENERAL:

Parameter Style GENERAL WITH NULLS:

Parameter style DB2SQL:



Notes:

1 On DB2 UDB for UWO, an *SQL-argument-ind-array* is used.

SQL-argument
: Each *SQL-argument* represents one input or output value defined when the routine was created.

  The following describes the use of each *SQL-argument*.

  - An IN parameter of a procedure is set by the database manager before invoking the program. The value of each of these arguments is taken from

the expression specified in the CALL to the procedure. It is assigned to the corresponding parameter definition in the CREATE PROCEDURE statement using storage assignment as described in "Assignments and Comparisons" on page 58.

These arguments are input only and any changes to these argument values made by the program are ignored upon return from the program.

- An OUT parameter of a procedure is set by the program before returning to the database manager. The program must return a value that corresponds to the data type and length of the result as specified when the procedure was created. See "Attributes of the Arguments Passed to a Routine Program" on page 660 for appropriate data type declarations.
- An INOUT parameter of a procedure behaves as both an IN and an OUT parameter and therefore follows both sets of rules described above. The database manager will set the argument before invoking the program.

SQL-argument-ind-array

There is an element in *SQL-argument-ind-array* for each *SQL-argument* passed to the program. *SQL-argument-ind-array* is an array of two-byte signed integers. The *n*th element of *SQL-argument-ind-array* corresponds to the *n*th *SQL-argument*. The elements of the array can be used by the program to determine if the corresponding *SQL-argument* is null or not.

The following describes the use of each *SQL-argument-ind-array* element.

- An IN parameter of a procedure is set by DB2 before invoking the program. It contains one of the following values:

  0 The procedure argument is present and not NULL.

  -1 The procedure argument value is NULL.

  The program should check every input argument's *SQL-argument-ind-array* element since any argument can be NULL.
- An OUT parameter of a procedure which must be set by the program before returning to the database manager. This argument is used by the program to indicate if the particular returned value is NULL:

  0 or positive
  The returned value is present and not NULL.

  negative
  The returned value is NULL.

  The program must set the *SQL-argument-ind-array* element of all output parameters. If the indicator value is other than -1 or -2, the returned value may not be the same as the value specified in the program.
- An INOUT parameter of a procedure behaves as both an IN and an OUT parameter and therefore follows both sets of rules described above.

SQL-argument-ind

There is an *SQL-argument-ind* for each *SQL-argument* passed to the program. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* and indicates whether the *SQL-argument* has a value or is NULL.

Each *SQL-argument-ind* is defined as a two-byte signed integer. The use of each *SQL-argument-ind* is the same as the use of each element of *SQL-argument-ind-array*.

G G G DB2 UDB for UWO uses an *SQL-argument-ind-array* for parameter style DB2SQL on procedures. See the description of *SQL-argument-ind-array* for details.

SQL-state
: This output argument is a CHAR(5) value that represents the SQLSTATE. This argument is passed in from the database manager with the initial value set to '00000' and can be set by the program as an SQLSTATE for the procedure. While normally the SQLSTATE is not set by the program, it can be used to return an error or warning to the database as follows:

  01Hxx
  : The program detected a warning situation. This results in an SQL warning. Here *xx* may be one of several possible strings.

  38xxx
  : The program detected an error situation. It results in an SQL error. Here *xxx* may be one of several possible strings.

  See Appendix E, "SQLSTATE Values—Common Return Codes" on page 539 for more information about valid SQLSTATEs that the program may use.

qualified-procedure-name
: This input argument is set by the database manager before invoking the program. It is a VARCHAR(517) value that contains the name of the procedure that is invoking the program. The format of the value in *qualified-procedure-name* is:

```
"schema-name"."procedure-name"
```

Note that any double quote character within the *schema-name* or *procedure-name* gets doubled. This argument is useful when the program is being used by multiple procedure definitions so that the program can distinguish which procedure is being invoked. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

specific-name
: This input argument is set by the database manager before invoking the program. It is a VARCHAR(128) value that contains the specific name of the procedure that is invoking the program. Like *qualified-procedure-name*, this parameter is useful when the routine code is being used by multiple procedure definitions so that the program can distinguish which definition is being invoked. See CREATE PROCEDURE for more information about *specific-name*. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

diagnostic-message
: This output argument is a VARCHAR(70) value that can be used by the program to send message text back when an SQLSTATE warning or error is returned by the program. It is initialized by the database manager to an empty string before invoking the program and may be set by the program with descriptive information. The *diagnostic-message* argument value is ignored by the database manager unless the *SQL-state* argument is set by the program to a value other than '00000'.

dbinfo
: This output argument is set by the database manager before invoking the program. It is only present if the CREATE PROCEDURE statement for the

program specifies the DBINFO keyword. The argument is a structure whose definition is described in "Database Information in External Routines (DBINFO)" on page 662. This argument is input only and any changes to the argument value made by the program are ignored by the database manager upon return from the program.

Parameter Passing for External Procedures Written in Java

The Java parameter style is the style specified by *Information technology - Database languages - SQL - Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)* ISO/IEC 9075-13:2002. When coding a Java method for an external procedure, the following conventions must be followed.

- The Java method must be a public void static (not instance) method.
- The parameters of the Java method must be a Java type that is equivalent to the SQL data type of the parameter (see Table 72 on page 661).
- The output parameters must be returned using single element arrays.
- If the procedure is defined with DYNAMIC RESULT SETS *n*, where *n* is greater than zero, the Java method signature must end with *n* parameters whose type is java.sql.ResultSet[ ]. All java.sql.ResultSets to be returned to the calling application must be assigned to the first element of the array representing their output parameter, and all ResultSets that are not being returned to the calling application need to be explicitly or implicitly closed before the procedure returns.

Consider an example of a procedure created with parameters of SQL types t1, t2, and t3, and t4 with external name 'jarfile.pname' (jarfile is the Java class name). The database manager will invoke the Java method with the expected Java signature:

```
public static void pname (T1 a, T2 b, T3 c, T4 d) { ... }
```

Where:

- pname is the Java method name
- T1 through T4 are the Java types that correspond to SQL types t1 through t4
- a, b, c, and d are arbitrary variable names for the arguments.

SQL NULL values are represented by uninitialized Java variables that are non-primitive types. The following primitive Java types do not support the SQL NULL value: short, int, long, float, double. If a null value is passed to a parameter that is a Java primitive type, an SQL error is returned.

Attributes of the Arguments Passed to a Routine Program

Table 72 on page 661 should be used to determine the appropriate type declarations for the parameters of the program associated with a routine. Each programming language supports different data types. The SQL data type is contained in the leftmost column of the table. Other columns in that row contain an indication of whether that data type is supported as a parameter type for a particular language. If the column contains a dash (-), the data type is not supported as a parameter type for that language.

150. In C, C++, or COBOL, a datetime value is passed to a routine using a string representation in the ISO format. For example, a TIME value is passed to routine as the string '12.58.01'. When returning a datetime value, any of the supported datetime string formats may be used. See "String Representations of Datetime Values" on page 49 for details.

Table 72. Data Type Mappings for Parameters

SQL Data Type	C and C++	COBOL	Java
SMALLINT	short	PIC S9(4) BINARY	short
INTEGER	long	PIC S9(9) BINARY	int
DECIMAL(p,s) or NUMERIC(p,s)	-	PIC S9(p-s)V9(s) PACKED-DECIMAL **Note:** Precision must not be greater than 18.	java.math.BigDecimal
REAL or FLOAT(p)	float	COMP-1	float
DOUBLE PRECISION or FLOAT or FLOAT(p)	double	COMP-2	double
CHARACTER(n)	char ... [n+1]	PIC X(n)	java.lang.String
CHAR(n) FOR BIT DATA	char ... [n+1]	PIC X(n)	-
VARCHAR(n)	char ... [n+1]	Varying-Length Character String (see "Character Host Variables (excluding CLOB)" on page 615)	java.lang.String
VARCHAR(n) FOR BIT DATA	VARCHAR structured form (see "Character Host Variables (excluding CLOB)" on page 598)	Varying-Length Character String (see "Character Host Variables (excluding CLOB)" on page 615.	-
GRAPHIC(n)	wchar_t ... [n+1]	PIC G(n) DISPLAY-1 or PIC N(n)	java.lang.String
VARGRAPHIC(n)	VARGRAPHIC structured form (see C chapter)	Varying-Length Graphic String (see "Graphic Host Variables (excluding DBCLOB)" on page 616)	java.lang.String
DATE[150]	char ... [11]	PIC X(10)	java.sql.Date
TIME[150]	char ... [9]	PIC X(8)	java.sql.Time
TIMESTAMP[150]	char ... [27]	PIC X(26)	java.sql.Timestamp
CLOB	CLOB structured form (see "Declaring a LOB Parameter" on page 662)	CLOB structured form (see "Declaring a LOB Parameter" on page 662)	-
BLOB	BLOB structured form (see "Declaring a LOB Parameter" on page 662)	BLOB structured form (see "Declaring a LOB Parameter" on page 662)	-
DBCLOB	DBCLOB structured form (see "Declaring a LOB Parameter" on page 662)	DBCLOB structured form (see "Declaring a LOB Parameter" on page 662)	-
distinct type	[151]	[151]	[151]

Table 72. Data Type Mappings for Parameters (continued)

SQL Data Type	C and C++	COBOL	Java
Indicator Variable	short	PIC S9(4) BINARY	Not applicable for Java

Declaring a LOB Parameter

The declaration of a LOB parameter for a routine written in C or COBOL requires a structure with a length and data fields.

- For a CLOB or BLOB in C, the following is an example declaration for a CLOB(64K) or BLOB(64K) parameter:

```
struct parm1_t
{
  unsigned long length;
  char          data[65536];
} parm1;
```

- For a DBCLOB in C, the following is an example declaration for a DBCLOB(64K) parameter:

```
struct parm2_t
{
  unsigned long length;
  wchar_t       data[65536];
} parm2;
```

- For a CLOB or BLOB in COBOL, the following is an example declaration for a CLOB(64K) or BLOB(64K) parameter:

```
01 LOB-PARM1.
   49 LOB-PARM1-LENGTH PIC 9(9) BINARY.
   49 LOB-PARM1-DATA PIC X(65536).
```

- For a DBCLOB in C, the following is an example declaration for a DBCLOB(64K) parameter:

```
01 DBCLOB-PARM2.
   49 DBCLOB-PARM2-LENGTH PIC 9(9) BINARY.
   49 DBCLOB-PARM2-DATA PIC G(8192) DISPLAY-1.
```

Database Information in External Routines (DBINFO)

Routines sometimes need access to information about the current server and where the routine is invoked. Table 73 contains a description of the relevant fields of the DBINFO structure which provide such information. Detailed information about the DBINFO structure can be found in the sqludf include file.

Table 73. DBINFO fields

Field	Data Type[152]	Description
Relational database name	VARCHAR(128)	The name of the current server.
Authorization ID	VARCHAR(128)	The execution time authorization ID.

151. A distinct type parameter is passed as the source type of the distinct type. Refer to the source type of the distinct type to determine the appropriate language type.

152. A data type of VARCHAR(n) in this table implies that there is a 2 byte length field followed by character string data. The character string may not be nul-terminated.

Table 73. DBINFO fields (continued)

Field	Data Type[152]	Description
Environment CCSID Information	structure (see "DBINFO Structure for C" on page 664 or "DBINFO Structure for COBOL" on page 665)	The CCSID information of the environment. See "CCSID Information in DBINFO" on page 664 for more details.
Schema name	VARCHAR(128)	Schema name of the target table where the function reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement. Otherwise blank.
Table name	VARCHAR(128)	Table name of the target table where the function reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement. Otherwise blank.
Column name	VARCHAR(128)	Column name of the target column where the function reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement. Otherwise blank.
Product information	CHAR(8)	Identifies the product on which the routine executes. The information has the form *pppvvrrm*, where: • *ppp* is: DSN for DB2 UDB for z/OS and OS/390 QSQ for DB2 UDB for iSeries SQL for DB2 UDB for UWO • *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 z/OS and OS/390, the value would be 'DSN07010'.

Table 73. DBINFO fields (continued)

Field	Data Type[152]	Description
Platform type	INTEGER	Identifies the operating system on which the program that invokes the routine runs. The value is one of these: • 0 Unknown • 1 OS/2 • 3 Windows • 4 AIX • 5 Windows NT • 6 HP-UX • 7 Solaris • 8 OS/390 or z/OS • 13 Siemens Nixdorf • 15 Windows 95 • 16 SCO Unix • 24 Linux/390 • 400 OS/400
Application identifier	pointer to character string	A unique identifier for the application invoking the routine.

CCSID Information in DBINFO

The environment CCSID information provided in DBINFO is presented in the form of 3 sets of 3 CCSIDs. Each set consists of an SBCS CCSID, a DBCS CCSID, and a mixed CCSID. The reason for 3 sets of CCSIDs is to allow representations of the different encoding schemes that are possible. Therefore the field following the sets of CCSIDs indicates which set is relevant. The environment CCSIDs provide the routine with information about the CCSID that is used See “Coded Character Sets and CCSIDs” on page 26 for more information on CCSIDs and codepages.

G G The meaning of these environment CCSIDs depends on the application server where the routine is executed.

G G
- On DB2 UDB for z/OS and OS/390, the environment CCSIDs are the CCSIDs for the table accessed in the containing SQL statement.

G G
- On DB2 UDB for iSeries, the environment CCSIDs are the CCSIDs associated with the job.

G G
- On DB2 UDB for UWO, the environment CCSIDs are the CCSIDs for the relational database.

DBINFO Structure for C

In C, the DBINFO structure and associated structure declarations are equivalent (but not necessarily identical) to the following.

```
#define   SQLUDF_MAX_IDENT_LEN    128    /* max length of identifier */
#define   SQLUDF_SH_IDENT_LEN       8    /* length of short identifier */

/*-------------------------------------------------------------------*/
/* Structure used for: Environment CCSID                             */
/*-------------------------------------------------------------------*/

SQL_STRUCTURE db2_cdpg
{
    struct db2_ccsids
```

```
    {
       unsigned long   db2_sbcs;
       unsigned long   db2_dbcs;
       unsigned long   db2_mixed;
    } db2_ccsids_t[3];

   unsigned long   db2_encoding_scheme;
   unsigned char reserved[8];
};
/*-----------------------------------------------------------------*/
/* encoding_scheme values for db2_cdpg.db2_encoding_scheme       */
/*-----------------------------------------------------------------*/
#define   SQLUDF_ASCII             0       /* ASCII              */
#define   SQLUDF_EBCDIC            1       /* EBCDIC             */
#define   SQLUDF_UNICODE           2       /* UNICODE            */

/*--------------------------------------------------------------*/
/* Structure used for: dbinfo.                                  */
/*--------------------------------------------------------------*/
SQL_STRUCTURE  sqludf_dbinfo
{
  unsigned short    dbnamelen;                      /* database name length    */
  unsigned char     dbname[SQLUDF_MAX_IDENT_LEN];  /* database name           */
  unsigned short    authidlen;                      /* authorization ID length */
  unsigned char     authid[SQLUDF_MAX_IDENT_LEN];  /* appl authorization ID   */
  struct db_cdpg    codepg;                         /* database code page      */
  unsigned short    tbschemalen;                    /* table schema name length*/
  unsigned char     tbschema[SQLUDF_MAX_IDENT_LEN];/* table schema name       */
  unsigned short    tbnamelen;                      /* table name length       */
  unsigned char     tbname[SQLUDF_MAX_IDENT_LEN];  /* table name              */
  unsigned short    colnamelen;                     /* column name length      */
  unsigned char     colname[SQLUDF_MAX_IDENT_LEN]; /* column name             */
  unsigned char     ver_rel[SQLUDF_SH_IDENT_LEN];  /* product information     */
  unsigned long     platform;                       /* platform type           */
  unsigned char     resd1[28];                      /* Reserved- for expansion */
  char             *appl_id;                        /* application identifier  */
  unsigned char     resd2[20];                      /* Reserved- for expansion */
};
```

DBINFO Structure for COBOL

In COBOL, the DBINFO structure and associated structure declarations are equivalent (but not necessarily identical) to the following.

```
      ***********************************************************************
      * Structure used for: dbinfo.
      ************************************************************************
       01 SQLUDF-DBINFO.
      * relational database name length
           05 DBNAMELEN              PIC 9(4) COMP-5.
      * reltional database name
           05 DBNAME                 PIC X(128).
      * authorization ID length
           05 AUTHIDLEN              PIC 9(4) COMP-5.
      * authorization ID
           05 AUTHID                 PIC X(128).
      * environment CCSID information
           05 CODEPG.
               10 CDPG.
                   15 ASCII-SBCS     PIC 9(9) COMP-5.
                   15 ASCII-DBCS     PIC 9(9) COMP-5.
                   15 ASCII-MIXED    PIC 9(9) COMP-5.
                   15 EBCDIC-SBCS    PIC 9(9) COMP-5.
                   15 EBCDIC-DBCS    PIC 9(9) COMP-5.
                   15 EBCDIC-MIXED   PIC 9(9) COMP-5.
                   15 UNICODE-SBCS    PIC 9(9) COMP-5.
                   15 UNICODE-DBCS    PIC 9(9) COMP-5.
```

```
                 15 UNICODE-MIXED   PIC 9(9) COMP-5.
                 15 ENCODING-SCHEME PIC 9(9) COMP-5.
                 15 RESERVED        PIC X(8).
* schema name length
     05 TBSCHEMALEN            PIC 9(4) COMP-5.
* schema name
     05 TBSCHEMA               PIC X(128).
* table name length
     05 TBNAMELEN              PIC 9(4) COMP-5.
* table name
     05 TBNAME                 PIC X(128).
* column name length
     05 COLNAMELEN             PIC 9(4) COMP-5.
* column name
     05 COLNAME                PIC X(128).
* product information
     05 VER-REL                PIC X(8).
* Reserved for expansion
     05 RESD0                  PIC X(2).
* platform type
     05 PLATFORM               PIC 9(9) COMP-5.
* Reserved for expansion
     05 RESD1                  PIC X(28).
* application identifier
     05 APPL-ID                USAGE IS POINTER.
* Reserved for expansion
     05 RESD2                  PIC X(20).
```

Scratch Pad in External Functions

External functions may need an area to save information between invocations. This is referred to as a *scratch pad*. A function is enabled to have a scratch pad by specifying the SCRATCHPAD keyword during CREATE FUNCTION (see "CREATE FUNCTION (External Scalar)" on page 314). Table 74 contains a description of the fields of the scratchpad structure. Detailed information about the scratchpad structure can be found in the sqludf include file.

Table 74. SCRATCHPAD fields

Field	Data Type	Description
Length of scratchpad	INTEGER	Length of the data field of the scratchpad.
Scratchpad area	CHARACTER(100)	The data area available for a scratchpad. Actual length of the scratchpad can exceed 100, but the structure definition in the sqludf include file defaults to 100.

The following is an example of a C declaration for a scratchpad with 150 bytes.

```
SQL_STRUCTURE  sqludf_scratchpad
{
  unsigned long  length;      /* length of scratchpad data       */
  char           data[150];   /* scratchpad data, init. to all \0 */
};
```

Appendix M. Sample Tables

The tables on the following pages are used in the examples that appear throughout this book. This appendix contains the following sample tables:

“ACT”
“CL_SCHED” on page 668
“DEPARTMENT” on page 668
“EMP_PHOTO” on page 668
“EMP_RESUME” on page 669
“EMPLOYEE” on page 669
“EMPPROJACT” on page 672
“IN_TRAY” on page 674
“ORG” on page 674
“PROJACT” on page 675
“PROJECT” on page 676.
“SALES” on page 678
“STAFF” on page 679

In these tables, a question mark (?) indicates a null value.

ACT

Name:	ACTNO	ACTKWD	ACTDESC
Type:	SMALLINT NOT NULL	CHAR(6) NOT NULL	VARCHAR(20) NOT NULL
Desc:	Account number	Account keyword	Account description
Values:	10	MANAGE	MANAGE/ADVISE
	20	ECOST	ESTIMATE COST
	30	DEFINE	DEFINE SPECS
	40	LEADPR	LEAD PROGRAM/DESIGN
	50	SPECS	WRITE SPECS
	60	LOGIC	DESCRIBE LOGIC
	70	CODE	CODE PROGRAMS
	80	TEST	TEST PROGRAMS
	90	ADMQS	ADM QUERY SYSTEM
	100	TEACH	TEACH CLASSES
	110	COURSE	DEVELOP COURSES
	120	STAFF	PERS AND STAFFING
	130	OPERAT	OPER COMPUTER SYS
	140	MAINT	MAINT SOFTWARE SYS
	150	ADMSYS	ADM OPERATING SYS
	160	ADMDB	ADM DATA BASES

© Copyright IBM Corp. 1982, 2002

Name:	ACTNO	ACTKWD	ACTDESC
	170	ADMDC	ADM DATA COMM
	180	DOC	DOCUMENT

CL_SCHED

Name:	CLASS_CODE	DAY	STARTING	ENDING
Type:	CHAR(7)	SMALLINT	TIME	TIME
Desc:	Class code (room:teacher)	Day # of 4 day schedule	Class start time	Class end time
Values:	042:BF	4	12:10 PM	02:00 PM
	553:MJA	1	10:30 AM	11:00 AM
	543:CWM	3	09:10 AM	10:30 AM
	778:RES	2	12:10 PM	02:00 PM
	044:HD	3	05:12 PM	06:00 PM

DEPARTMENT

Name:	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
Type:	CHAR(3) NOT NULL	VARCHAR(29) NOT NULL	CHAR(6)	CHAR(3) NOT NULL	CHAR(16)
Desc:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) to which this department reports	Name of the remote location
Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	?
	B01	PLANNING	000020	A00	?
	C01	INFORMATION CENTER	000030	A00	?
	D01	DEVELOPMENT CENTER	?	A00	?
	D11	MANUFACTURING SYSTEMS	000060	D01	?
	D21	ADMINISTRATION SYSTEMS	000070	D01	?
	E01	SUPPORT SERVICES	000050	A00	?
	E11	OPERATIONS	000090	E01	?
	E21	SOFTWARE SUPPORT	000100	E01	?

EMP_PHOTO

Name:	EMPNO	PHOTO_FORMAT	PICTURE
Type:	CHAR(6) NOT NULL	VARCHAR(10) NOT NULL	BLOB(100K)
Desc:	Employee number	Photograph format	Photograph
Values:	000130	bitmap	Figure 16 on page 680

Name:	EMPNO	PHOTO_FORMAT	PICTURE
	000130	gif	Figure 16 on page 680
	000140	bitmap	Figure 18 on page 682
	000140	gif	Figure 18 on page 682
	000150	bitmap	Figure 20 on page 684
	000150	gif	Figure 20 on page 684
	000190	bitmap	Figure 22 on page 686
	000190	gif	Figure 22 on page 686

EMP_RESUME

Name:	EMPNO	RESUME_FORMAT	RESUME
Type:	CHAR(6) NOT NULL	VARCHAR(10) NOT NULL	CLOB(5K)
Desc:	Employee number	Resume format	Resume
Values:	000130	ascii	Figure 17 on page 681
	000130	html	Figure 17 on page 681
	000140	ascii	Figure 19 on page 683
	000140	html	Figure 19 on page 683
	000150	ascii	Figure 21 on page 685
	000150	html	Figure 21 on page 685
	000190	ascii	Figure 23 on page 687
	000190	html	Figure 23 on page 687

EMPLOYEE

Names:	EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
Type:	CHAR(6) NOT NULL	VARCHAR(12) NOT NULL	CHAR(1) NOT NULL	VARCHAR(15) NOT NULL	CHAR(3)	CHAR(4)	DATE
Desc:	Employee number	First name	Middle initial	Last name	Department (DEPTNO) in which the employee works	Phone number	Date of hire

JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
CHAR(8)	SMALLINT NOT NULL	CHAR(1)	DATE	DECIMAL(9,2)	DECIMAL(9,2)	DECIMAL(9,2)
Job	Number of years of formal education	Sex (M male, F female)	Date of birth	Yearly salary	Yearly bonus	Yearly commission

See the following page for the values in the EMPLOYEE table.

EMPNO	FIRSTNME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420

EMPNO	FIRSTNME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FILEREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FILEREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FILEREP	16	M	1926-05-17	23840	500	1907

EMPPROJACT

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
Type:	CHAR(6) NOT NULL	CHAR(6) NOT NULL	SMALLINT NOT NULL	DECIMAL(5,2)	DATE	DATE
Desc:	Employee number	Project number	Activity number	Proportion of employee's full time to be spent on project	Date activity starts	Date activity ends
Values:	000010	AD3100	10	.50	1982-01-01	1982-07-01
	000070	AD3110	10	1.00	1982-01-01	1983-02-01
	000230	AD3111	60	1.00	1982-01-01	1982-03-15
	000230	AD3111	60	.50	1982-03-15	1982-04-15
	000230	AD3111	70	.50	1982-03-15	1982-10-15
	000230	AD3111	80	.50	1982-04-15	1982-10-15
	000230	AD3111	180	.50	1982-10-15	1983-01-01
	000240	AD3111	70	1.00	1982-02-15	1982-09-15
	000240	AD3111	80	1.00	1982-09-15	1983-01-01
	000250	AD3112	60	1.00	1982-01-01	1982-02-01
	000250	AD3112	60	.50	1982-02-01	1982-03-15
	000250	AD3112	60	1.00	1983-01-01	1983-02-01
	000250	AD3112	70	.50	1982-02-01	1982-03-15
	000250	AD3112	70	1.00	1982-03-15	1982-08-15
	000250	AD3112	70	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.50	1982-10-15	1982-12-01
	000250	AD3112	180	.50	1982-08-15	1983-01-01
	000260	AD3113	70	.50	1982-06-15	1982-07-01
	000260	AD3113	70	1.00	1982-07-01	1983-02-01
	000260	AD3113	80	1.00	1982-01-01	1982-03-01
	000260	AD3113	80	.50	1982-03-01	1982-04-15
	000260	AD3113	180	.50	1982-03-01	1982-04-15
	000260	AD3113	180	1.00	1982-04-15	1982-06-01
	000260	AD3113	180	1.00	1982-06-01	1982-07-01
	000270	AD3113	60	.50	1982-03-01	1982-04-01
	000270	AD3113	60	1.00	1982-04-01	1982-09-01
	000270	AD3113	60	.25	1982-09-01	1982-10-15
	000270	AD3113	70	.75	1982-09-01	1982-10-15
	000270	AD3113	70	1.00	1982-10-15	1983-02-01
	000270	AD3113	80	1.00	1982-01-01	1982-03-01
	000270	AD3113	80	.50	1982-03-01	1982-04-01
	000030	IF1000	10	.50	1982-06-01	1983-01-01
	000130	IF1000	90	1.00	1982-10-01	1983-01-01

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000130	IF1000	100	.50	1982-10-01	1983-01-01
	000140	IF1000	90	.50	1982-10-01	1983-01-01
	000030	IF2000	10	.50	1982-01-01	1983-01-01
	000140	IF2000	100	1.00	1982-01-01	1982-03-01
	000140	IF2000	100	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-10-01	1983-01-01
	000010	MA2100	10	.50	1982-01-01	1982-11-01
	000110	MA2100	20	1.00	1982-01-01	1983-03-01
	000010	MA2110	10	1.00	1982-01-01	1983-02-01
	000200	MA2111	50	1.00	1982-01-01	1982-06-15
	000200	MA2111	60	1.00	1982-06-15	1983-02-01
	000220	MA2111	40	1.00	1982-01-01	1983-02-01
	000150	MA2112	60	1.00	1982-01-01	1982-07-15
	000150	MA2112	180	1.00	1982-07-15	1983-02-01
	000170	MA2112	60	1.00	1982-01-01	1983-06-01
	000170	MA2112	70	1.00	1982-06-01	1983-02-01
	000190	MA2112	70	1.00	1982-01-01	1982-10-01
	000190	MA2112	80	1.00	1982-10-01	1983-10-01
	000160	MA2113	60	1.00	1982-07-15	1983-02-01
	000170	MA2113	80	1.00	1982-01-01	1983-02-01
	000180	MA2113	70	1.00	1982-04-01	1982-06-15
	000210	MA2113	80	.50	1982-10-01	1983-02-01
	000210	MA2113	180	.50	1982-10-01	1983-02-01
	000050	OP1000	10	.25	1982-01-01	1983-02-01
	000090	OP1010	10	1.00	1982-01-01	1983-02-01
	000280	OP1010	130	1.00	1982-01-01	1983-02-01
	000290	OP1010	130	1.00	1982-01-01	1983-02-01
	000300	OP1010	130	1.00	1982-01-01	1983-02-01
	000310	OP1010	130	1.00	1982-01-01	1983-02-01
	000050	OP2010	10	.75	1982-01-01	1983-02-01
	000100	OP2010	10	1.00	1982-01-01	1983-02-01
	000320	OP2011	140	.75	1982-01-01	1983-02-01
	000320	OP2011	150	.25	1982-01-01	1983-02-01
	000330	OP2012	140	.25	1982-01-01	1983-02-01
	000330	OP2012	160	.75	1982-01-01	1983-02-01
	000340	OP2013	140	.50	1982-01-01	1983-02-01
	000340	OP2013	170	.50	1982-01-01	1983-02-01
	000020	PL2100	30	1.00	1982-01-01	1982-09-15

IN_TRAY

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
Type:	TIMESTAMP	CHAR(8)	CHAR(64)	VARCHAR(3000)
Desc:	Date and time received	User id of person sending note	Brief description	The note
Values:	1988-12-25-17.12.30.000000	BADAMSON	FWD: Fantastic year! 4th Quarter Bonus.	To: JWALKER Cc: QUINTANA, NICHOLLS Jim, Looks like our hard work has paid off. I have some good beer in the fridge if you want to come over to celebrate a bit. Delores and Heather, are you interested as well? Bruce Subject: FWD: Fantastic year! 4th Quarter Bonus. To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
	1988-12-23-08.53.58.000000	ISTERN	FWD: Fantastic year! 4th Quarter Bonus.	To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
	1988-12-22-14.07.21.136421	CHAAS	Fantastic year! 4th Quarter Bonus.	To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

ORG

Name:	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
Type:	SMALLINT NOT NULL	VARCHAR(14)	SMALLINT	VARCHAR(10)	VARCHAR(13)
Desc:	Department number	Department name	Manager number	Division	Location
Values:	10	Head Office	160	Corporate	New York
	15	New England	50	Eastern	Boston
	20	Mid Atlantic	10	Eastern	Washington
	38	South Atlantic	30	Eastern	Atlanta

Name:	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
	42	Great Lakes	100	Midwest	Chicago
	51	Plains	140	Midwest	Dallas
	66	Pacific	270	Western	San Francisco
	84	Mountain	290	Western	Denver

PROJACT

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
Type:	CHAR(6) NOT NULL	SMALLINT NOT NULL	DECIMAL(5,2)	DATE NOT NULL	DATE
Desc:	Project number	Activity number	Activity staffing	Activity start	Activity end
Values:	AD3100	10	?	1982-01-01	?
	AD3110	10	?	1982-01-01	?
	AD3111	60	?	1982-01-01	?
	AD3111	60	?	1982-03-15	?
	AD3111	70	?	1982-03-15	?
	AD3111	80	?	1982-04-15	?
	AD3111	180	?	1982-10-15	?
	AD3111	70	?	1982-02-15	?
	AD3111	80	?	1982-09-15	?
	AD3112	60	?	1982-01-01	?
	AD3112	60	?	1982-02-01	?
	AD3112	60	?	1983-01-01	?
	AD3112	70	?	1982-02-01	?
	AD3112	70	?	1982-03-15	?
	AD3112	70	?	1982-08-15	?
	AD3112	80	?	1982-08-15	?
	AD3112	80	?	1982-10-15	?
	AD3112	180	?	1982-08-15	?
	AD3113	70	?	1982-06-15	?
	AD3113	70	?	1982-07-01	?
	AD3113	80	?	1982-01-01	?
	AD3113	80	?	1982-03-01	?
	AD3113	180	?	1982-03-01	?
	AD3113	180	?	1982-04-15	?
	AD3113	180	?	1982-06-01	?
	AD3113	60	?	1982-03-01	?
	AD3113	60	?	1982-04-01	?
	AD3113	60	?	1982-09-01	?
	AD3113	70	?	1982-09-01	?
	AD3113	70	?	1982-10-15	?

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
	IF1000	10	?	1982-06-01	?
	IF1000	90	?	1982-10-01	?
	IF1000	100	?	1982-10-01	?
	IF2000	10	?	1982-01-01	?
	IF2000	100	?	1982-01-01	?
	IF2000	100	?	1982-03-01	?
	IF2000	110	?	1982-03-01	?
	IF2000	110	?	1982-10-01	?
	MA2100	10	?	1982-01-01	?
	MA2100	20	?	1982-01-01	?
	MA2110	10	?	1982-01-01	?
	MA2111	50	?	1982-01-01	?
	MA2111	60	?	1982-06-15	?
	MA2111	40	?	1982-01-01	?
	MA2112	60	?	1982-01-01	?
	MA2112	180	?	1982-07-15	?
	MA2112	70	?	1982-06-01	?
	MA2112	70	?	1982-01-01	?
	MA2112	80	?	1982-10-01	?
	MA2113	60	?	1982-07-15	?
	MA2113	80	?	1982-01-01	?
	MA2113	70	?	1982-04-01	?
	MA2113	80	?	1982-10-01	?
	MA2113	180	?	1982-10-01	?
	OP1000	10	?	1982-01-01	?
	OP1010	10	?	1982-01-01	?
	OP1010	130	?	1982-01-01	?
	OP2010	10	?	1982-01-01	?
	OP2011	140	?	1982-01-01	?
	OP2011	150	?	1982-01-01	?
	OP2012	140	?	1982-01-01	?
	OP2012	160	?	1982-01-01	?
	OP2013	140	?	1982-01-01	?
	OP2013	170	?	1982-01-01	?
	PL2100	30	?	1982-01-01	?

PROJECT

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
Type:	CHAR(6) NOT NULL	VARCHAR(24) NOT NULL	CHAR(3) NOT NULL	CHAR(6) NOT NULL	DECIMAL(5,2)	DATE	DATE	CHAR(6)

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
Desc:	Project number	Project name	Department responsible	Employee responsible	Estimated mean staffing	Estimated start date	Estimated end date	Major project, for a subproject
Values:	AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	?
	AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
	AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110
	AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
	AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
	IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	?
	IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	?
	MA2100	WELD LINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	?
	MA2110	W L PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
	MA2111	W L PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110
	MA2112	W L ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110
	MA2113	W L PROD CONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
	OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	?
	OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
	OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	?
	OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
	OP2011	SCP SYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
	OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
	OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
	PL2100	WELD LINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

SALES

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
Type:	DATE	VARCHAR(15)	VARCHAR(15)	INTEGER
Desc:	Date	Sales person	Region	Sales amount
Values:	1995-12-31	LUCCHESSI	Ontario-South	1
	1995-12-31	LEE	Ontario-South	3
	1995-12-31	LEE	Quebec	1
	1995-12-31	LEE	Manitoba	2
	1995-12-31	GOUNOT	Quebec	1
	1996-03-29	LUCCHESSI	Ontario-South	3
	1996-03-29	LUCCHESSI	Quebec	1
	1996-03-29	LEE	Ontario-South	2
	1996-03-29	LEE	Ontario-North	2
	1996-03-29	LEE	Quebec	3
	1996-03-29	LEE	Manitoba	5
	1996-03-29	GOUNOT	Ontario-South	3
	1996-03-29	GOUNOT	Quebec	1
	1996-03-29	GOUNOT	Manitoba	7
	1996-03-30	LUCCHESSI	Ontario-South	1
	1996-03-30	LUCCHESSI	Quebec	2
	1996-03-30	LUCCHESSI	Manitoba	1
	1996-03-30	LEE	Ontario-South	7
	1996-03-30	LEE	Ontario-North	3
	1996-03-30	LEE	Quebec	7
	1996-03-30	LEE	Manitoba	4
	1996-03-30	GOUNOT	Ontario-South	2
	1996-03-30	GOUNOT	Quebec	18
	1996-03-30	GOUNOT	Manitoba	1
	1996-03-31	LUCCHESSI	Manitoba	1
	1996-03-31	LEE	Ontario-South	14
	1996-03-31	LEE	Ontario-North	3
	1996-03-31	LEE	Quebec	7
	1996-03-31	LEE	Manitoba	3
	1996-03-31	GOUNOT	Ontario-South	2
	1996-03-31	GOUNOT	Quebec	1
	1996-04-01	LUCCHESSI	Ontario-South	3
	1996-04-01	LUCCHESSI	Manitoba	1
	1996-04-01	LEE	Ontario-South	8
	1996-04-01	LEE	Ontario-North	?
	1996-04-01	LEE	Quebec	8
	1996-04-01	LEE	Manitoba	9

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
	1996-04-01	GOUNOT	Ontario-South	3
	1996-04-01	GOUNOT	Ontario-North	1
	1996-04-01	GOUNOT	Quebec	3
	1996-04-01	GOUNOT	Manitoba	7

STAFF

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	SMALLINT NOT NULL	VARCHAR(9)	SMALLINT	CHAR(5)	SMALLINT	DECIMAL(7,2)	DECIMAL(7,2)
Desc:	Staff ID	Name	Department	Job	Years of service	Salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	?
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	?
	40	O'Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	?
	60	Quigley	38	Sales	?	16808.30	650.25
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	?	13504.60	128.20
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	?
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	?	12954.75	180.00
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	?
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	?
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	?	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	?
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	?
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	?
	270	Lea	66	Mgr	9	18555.50	?
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	?

Sample Tables

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00

Sample Files With BLOB and CLOB Data Type

This section shows the data found in the EMP_PHOTO files (pictures of employees) and EMP_RESUME files (resumes of employees).

Quintana Photo

Figure 16. Dolores M. Quintana

Quintana Resume

Resume: Dolores M. Quintana

Personal Information

Address:	1150 Eglinton Ave Mellonville, Idaho 83725
Phone:	(208) 555-9933
Birthdate:	September 15, 1925
Sex:	Female
Marital Status:	Married
Height:	5′2″
Weight:	120 lbs.

Department Information

Employee Number:	000130
Dept Number:	C01
Manager:	Sally Kwan
Position:	Analyst
Phone:	(208) 555-4578
Hire Date:	1971-07-28

Education

1965	Math and English, B.A. Adelphi University
1960	Dental Technician Florida Institute of Technology

Work History

10/91 - present	Advisory Systems Analyst Producing documentation tools for engineering department.
12/85 - 9/91	Technical Writer, Writer, text programmer, and planner.
1/79 - 11/85	COBOL Payroll Programmer Writing payroll programs for a diesel fuel company.

Interests

- Cooking
- Reading
- Sewing
- Remodeling

Figure 17. Dolores M. Quintana

Nicholls Photo

Figure 18. Heather A. Nicholls

Nicholls Resume

Resume: Heather A. Nicholls

Personal Information

Address:	844 Don Mills Ave Mellonville, Idaho 83734
Phone:	(208) 555-2310
Birthdate:	January 19, 1946
Sex:	Female
Marital Status:	Single
Height:	5′8″
Weight:	130 lbs.

Department Information

Employee Number:	000140
Dept Number:	C01
Manager:	Sally Kwan
Position:	Analyst
Phone:	(208) 555-1793
Hire Date:	1976-12-15

Education

1972	Computer Engineering, Ph.D. University of Washington
1969	Music and Physics, M.A. Vassar College

Work History

2/83 - present	Architect, OCR Development Designing the architecture of OCR products.
12/76 - 1/83	Text Programmer Optical character recognition (OCR) programming in PL/I.
9/72 - 11/76	Punch Card Quality Analyst Checking punch cards met quality specifications.

Interests

- Model railroading
- Interior decorating
- Embroidery
- Knitting

Figure 19. Heather A. Nicholls

Adamson Photo

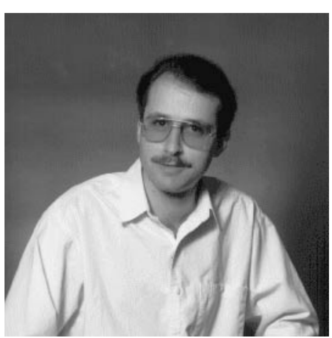


Figure 20. Bruce Adamson

Adamson Resume

Resume: Bruce Adamson

Personal Information

Address:	3600 Steeles Ave Mellonville, Idaho 83757
Phone:	(208) 555-4489
Birthdate:	May 17, 1947
Sex:	Male
Marital Status:	Married
Height:	6′0″
Weight:	175 lbs.

Department Information

Employee Number:	000150
Dept Number:	D11
Manager:	Irving Stern
Position:	Designer
Phone:	(208) 555-4510
Hire Date:	1972-02-12

Education

1971	Environmental Engineering, M.Sc. Johns Hopkins University
1968	American History, B.A. Northwestern University

Work History

8/79 - present	Neural Network Design Developing neural networks for machine intelligence products.
2/72 - 7/79	Robot Vision Development Developing rule-based systems to emulate sight.
9/71 - 1/72	Numerical Integration Specialist Helping bank systems communicate with each other.

Interests

- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

Figure 21. Bruce Adamson

Walker Photo

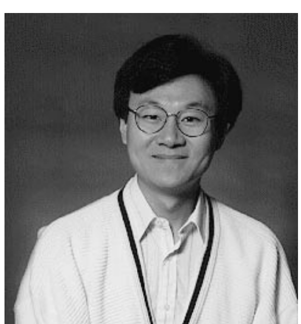


Figure 22. James H. Walker

Walker Resume

Resume: James H. Walker

Personal Information

Address:	3500 Steeles Ave Mellonville, Idaho 83757
Phone:	(208) 555-7325
Birthdate:	June 25, 1952
Sex:	Male
Marital Status:	Single
Height:	5′11″
Weight:	166 lbs.

Department Information

Employee Number:	000190
Dept Number:	D11
Manager:	Irving Stern
Position:	Designer
Phone:	(208) 555-2986
Hire Date:	1974-07-26

Education

1974	Computer Studies, B.Sc. University of Massachusetts
1972	Linguistic Anthropology, B.A. University of Toronto

Work History

6/87 - present	Microcode Design Optimizing algorithms for mathematical functions.
4/77 - 5/87	Printer Technical Support Installing and supporting laser printers.
9/74 - 3/77	Maintenance Programming Patching assembly language compiler for mainframes.

Interests

- Wine tasting
- Skiing
- Swimming
- Dancing

Figure 23. James H. Walker

Appendix N. Terminology Differences

Some terminology used in the ANSI and ISO standards differs from the terminology used in this book and other product books. The following tables cross-reference terms in the SQL 1999 Core standard to DB2 UDB SQL terms.

Table 75. ANSI/ISO Term to DB2 UDB SQL Term Cross-Reference

ANSI/ISO Term	DB2 UDB SQL Term
literal	constant
comparison predicate	basic predicate
comparison predicate subquery	subquery in a basic predicate
degree of table/cursor	number of items in a select list
grouped table	result table created by a group-by or having clause
grouped view	result view created by a group-by or having clause
grouping column	column in a group-by clause
outer reference	correlated reference
query expression	fullselect
query specification	subselect
result specification	result
set function	column function
table expression	►──from─clause──┬──────────────┬──────► └─where─clause─┘ ►──┬─────────────────┬──┬───────────────┬──► └─group─by─clause─┘ └─having─clause─┘
target specification	host variable followed by an indicator variable
transaction	logical unit of work or unit of work
value expression	arithmetic expression

Table 76. DB2 UDB SQL Term to ANSI/ISO Term Cross-Reference

DB2 UDB SQL Term	ANSI/ISO Term
arithmetic expression	value expression
basic predicate	comparison predicate
column function	set function
column in a group-by clause	grouping column
correlated reference	outer reference
►──from─clause──┬──────────────┬──────► └─where─clause─┘ ►──┬─────────────────┬──┬───────────────┬──► └─group─by─clause─┘ └─having─clause─┘	table expression
fullselect	query expression
host variable followed by an indicator variable	target specification
logical unit of work or unit of work	transaction

© Copyright IBM Corp. 1982, 2002

Table 76. DB2 UDB SQL Term to ANSI/ISO Term Cross-Reference (continued)

DB2 UDB SQL Term	ANSI/ISO Term
	direct SQL
number of items in a select list	degree of table/cursor
result	result specification
result table created by a group-by or having clause	grouped table
result view created by a group-by or having clause	grouped view
subquery in a basic predicate	comparison predicate subquery
subselect	query specification
subselect or fullselect in parenthesis	query term

Reserved Schema Names and Reserved Words

This appendix describes the restrictions of certain names used by the database manager. In some cases, names are reserved and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs though not prevented by the database manager.

Reserved Schema Names

The following schema names are reserved:

- QSYS2
- SYSCAT
- SYSFUN
- SYSIBM
- SYSPROC
- SYSSTAT
- SYSTEM

In addition, it is strongly recommended that schema names never begin with the Q prefix or SYS prefix, as Q and SYS are by convention used to indicate an area reserved by the system.

It is also recommended not to use SESSION as a schema name.

Reserved Words

The DB2 Universal Database reserved words are:

```
ADD             DETERMINISTIC  LEAVE          RESTART
AFTER           DISALLOW       LEFT           RESTRICT
ALIAS           DISCONNECT     LIKE           RESULT
ALL             DISTINCT       LINKTYPE       RESULT_SET_LOCATOR
ALLOCATE        DO             LOCAL          RETURN
ALLOW           DOUBLE         LOCALE         RETURNS
ALTER           DROP           LOCATOR        REVOKE
AND             DSNHATTR       LOCATORS       RIGHT
ANY             DSSIZE         LOCK           ROLLBACK
APPLICATION     DYNAMIC        LOCKMAX        ROUTINE
AS              EACH           LOCKSIZE       ROW
ASSOCIATE       EDITPROC       LONG           ROWS
ASUTIME         ELSE           LOOP           RRN
AUDIT           ELSEIF         MAXVALUE       RUN
AUTHORIZATION   ENCODING       MICROSECOND    SAVEPOINT
AUX             END            MICROSECONDS   SCHEMA
AUXILIARY       END-EXEC       MINUTE         SCRATCHPAD
BEFORE          END-EXEC1      MINUTES        SECOND
BEGIN           ERASE          MINVALUE       SECONDS
BETWEEN         ESCAPE         MODE           SECQTY
BINARY          EXCEPT         MODIFIES       SECURITY
BUFFERPOOL      EXCEPTION      MONTH          SELECT
BY              EXCLUDING      MONTHS         SENSITIVE
CACHE           EXECUTE        NEW            SET
CALL            EXISTS         NEW_TABLE      SIGNAL
CALLED          EXIT           NO             SIMPLE
CAPTURE         EXTERNAL       NOCACHE        SOME
CARDINALITY     FENCED         NOCYCLE        SOURCE
```

© Copyright IBM Corp. 1982, 2002

CASCADED
CASE
CAST
CCSID
CHAR
CHARACTER
CHECK
CLOSE
CLUSTER
COLLECTION
COLLID
COLUMN
COMMENT
COMMIT
CONCAT
CONDITION
CONNECT
CONNECTION
CONSTRAINT
CONTAINS
CONTINUE
COUNT
COUNT_BIG
CREATE
CROSS
CURRENT
CURRENT_DATE
CURRENT_LC_CTYPE
CURRENT_PATH
CURRENT_SERVER
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIMEZONE
CURRENT_USER
CURSOR
CYCLE
DATA
DATABASE
DAY
DAYS
DB2GENERAL
DB2GENRL
DB2SQL
DBINFO
DECLARE
DEFAULT
DEFAULTS
DEFINITION
DELETE
DESCRIPTOR
FETCH
FIELDPROC
FILE
FINAL
FOR
FOREIGN
FREE
FROM
FULL
FUNCTION
GENERAL
GENERATED
GET
GLOBAL
GO
GOTO
GRANT
GRAPHIC
GROUP
HANDLER
HAVING
HOLD
HOUR
HOURS
IDENTITY
IF
IMMEDIATE
IN
INCLUDING
INCREMENT
INDEX
INDICATOR
INHERIT
INNER
INOUT
INSENSITIVE
INSERT
INTEGRITY
INTO
IS
ISOBID
ISOLATION
ITERATE
JAR
JAVA
JOIN
KEY
LABEL
LANGUAGE
LC_CTYPE
NODENAME
NODENUMBER
NOMAXVALUE
NOMINVALUE
NOORDER
NOT
NULL
NULLS
NUMPARTS
OBID
OF
OLD
OLD_TABLE
ON
OPEN
OPTIMIZATION
OPTIMIZE
OPTION
OR
ORDER
OUT
OUTER
OVERRIDING
PACKAGE
PARAMETER
PART
PARTITION
PATH
PIECESIZE
PLAN
POSITION
PRECISION
PREPARE
PRIMARY
PRIQTY
PRIVILEGES
PROCEDURE
PROGRAM
PSID
QUERYNO
READ
READS
RECOVERY
REFERENCES
REFERENCING
RELEASE
RENAME
REPEAT
RESET
RESIGNAL
SPECIFIC
SQL
SQLID
STANDARD
START
STATIC
STAY
STOGROUP
STORES
STYLE
SUBPAGES
SUBSTRING
SYNONYM
SYSFUN
SYSIBM
SYSPROC
SYSTEM
TABLE
TABLESPACE
THEN
TO
TRANSACTION
TRIGGER
TRIM
TYPE
UNDO
UNION
UNIQUE
UNTIL
UPDATE
USAGE
USER
USING
VALIDPROC
VALUES
VARIABLE
VARIANT
VCAT
VIEW
VOLUMES
WHEN
WHERE
WHILE
WITH
WLM
WRITE
YEAR
YEARS

The ISO/ANSI SQL99 reserved words that are not in the list of DB2 Universal Database reserved words are:

ABSOLUTE
ACTION
ADMIN
AGGREGATE
ARE
ARRAY
ASC
ASSERTION
AT
BIT
BLOB
BOOLEAN
BOTH
DESCRIBE
DESTROY
DESTRUCTOR
DIAGNOSTICS
DICTIONARY
DOMAIN
EQUALS
EVERY
EXEC
FALSE
FIRST
FLOAT
FOUND
MODULE
NAMES
NATIONAL
NATURAL
NCHAR
NCLOB
NEXT
NONE
NUMERIC
OBJECT
OFF
ONLY
OPERATION
SESSION
SESSION_USER
SETS
SIZE
SMALLINT
SPACE
SPECIFICTYPE
SQLEXCEPTION
SQLSTATE
SQLWARNING
STATE
STATEMENT
STRUCTURE

```
BREADTH        GROUPING        ORDINALITY  SYSTEM_USER
CASCADE        HOST            OUTPUT      TEMPORARY
CATALOG        IGNORE          PAD         TERMINATE
CLASS          INITIALIZE      PARAMETERS  THAN
CLOB           INITIALLY       PARTIAL     TIME
COLLATE        INPUT           POSTFIX     TIMESTAMP
COLLATION      INT             PREFIX      TIMEZONE_HOUR
COMPLETION     INTEGER         PREORDER    TIMEZONE_MINUTE
CONSTRAINTS    INTERSECT       PRESERVE    TRAILING
CONSTRUCTOR    INTERVAL        PRIOR       TRANSLATION
CORRESPONDING  LARGE           PUBLIC      TREAT
CUBE           LAST            REAL        TRUE
CURRENT_ROLE   LATERAL         RECURSIVE   UNDER
DATE           LEADING         REF         UNKNOWN
DEALLOCATE     LESS            RELATIVE    UNNEST
DEC            LEVEL           ROLE        VALUE
DECIMAL        LIMIT           ROLLUP      VARCHAR
DEFERRABLE     LOCALTIME       SCOPE       VARYING
DEFERRED       LOCALTIMESTAMP  SCROLL      WHENEVER
DEPTH          MAP             SEARCH      WITHOUT
DEREF          MATCH           SECTION     WORK
DESC           MODIFY          SEQUENCE    ZONE
```

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Trademarks and Service Marks

The following terms are trademarks or service marks of the IBM Corporation in the United States and/or other countries:

AIX
CICS
DATABASE 2
DB2
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
iSeries
IBM
OS/2
OS/390
OS/400
z/OS

HP-UX is a trademark of Hewlett-Packard.

Java, JDBC, and Solaris are trademarks of Sun Microsystems, Inc.

Microsoft, Windows, Windows NT are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries, or both and is licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

© Copyright IBM Corp. 1982, 2002

Index

Special characters

_ (underscore) in LIKE predicate 120
: (colon) 86, 475
  C 597
  COBOL 612
  Java 632
  REXX 648
? (question mark)
  EXECUTE statement 401, 429
  PREPARE statement 434
/ (divide) operator 96
(and) 248
* (asterisk) 136
  in COUNT 135
  in subselect 235
* (multiply) operator 96
– (subtract) operator 96
% (percent) in LIKE predicate 120
> (greater than) operator 113, 114
>= (greater than or equal to) operator 113, 114
< (less than) operator 113, 114
<> (not equal to) operator 113, 114
<= (less than or equal to) operator 113, 114
+ (add) operator 96
= (equal to) operator
  in predicate 113, 114
  in UPDATE statement 467

A

ABS function 143
access plan 9
ACOS function 144
ACT sample table 667
administrative authority
  description 27
alias
  CREATE ALIAS statement 303
  description 8, 40
  dropping 396
ALIAS clause
  COMMENT statement 290
  DROP statement 396
alias-name
  description 32
  in COMMENT statement 289
  in DROP statement 396
ALL
  clause of RELEASE statement 440
  clause of subselect 235
  keyword
    AVG function 134
    column functions 133
    MAX function 137
    MIN function 138
    SUM function 140
  quantified predicate 114
ALL clause
  keyword
    COUNT_BIG function 136
    STDDEV function 139
    VAR function 141
    VARIANCE function 141
ALL PRIVILEGES clause
  GRANT (Table or View Privileges) statement 419
  REVOKE (Table and View Privileges) statement 452
ALL SQL
  clause of RELEASE statement 440
ALLOW PARALLEL clause
  CREATE FUNCTION statement 321
ALTER clause
  GRANT (Table or View Privileges) statement 419
  REVOKE (Table and View Privileges) statement 452
ALTER TABLE statement 267
ambiguous reference 82
AND
  operator in search condition 126
  truth table 126
ANY quantified predicate 114
application
  SQLJ support 627
application process, definition of 11
application program
  coding SQL statements
    C 593
    COBOL 609
    Java 627
    REXX 643
  concurrency 11
  SQLCA 525
  SQLDA 529
application requester 16, 520
application server 16, 520
application-directed access
  CONNECT (Type 2) statement 300
  mixed environment 516
arguments of COALESCE
  result data type 68
arithmetic expression
  equivalent term 689
arithmetic operators 96
AS clause of CREATE VIEW statement 376
AS LOCATOR clause
  CREATE FUNCTION statement 316
ASC clause
  CREATE INDEX statement 339
  select-statement 252
ASIN function 145
assignment
  C NUL-terminated strings 61
  datetime values 62
  distinct type 63
  LOB Locators 64
assignment *(continued)*
  mixed strings 61
  numeric 59
  operation in SQL 58
  retrieval 61
  storage 60
  string 60
assignment-clause
  in UPDATE statement 467
asterisk
  COUNT function 135
  multiply operator 96
  subselect 235
asterisk (*)
  in COUNT_BIG function 136
ATAN function 146
ATAN2 function 147
authorization
  description 27
  ID 41
  name 32
  privileges 28
  to create in a schema 28
authorization-name
  description 32
  in GRANT (Distinct Type Privileges) statement 410
  in GRANT (Function or Procedure Privileges) statement 412, 415
  in GRANT (package privileges) statement 416
  in GRANT (Table or View Privileges) statement 418, 420
  in REVOKE (Distinct Type Privileges) statement 444
  in REVOKE (Function and Procedure Privileges) statement 447, 449
  in REVOKE (package privileges) statement 450
  in REVOKE (Table and View Privileges) statement 452, 453
  length 36
AVG function 134

B

base table 3
basic operations in SQL 58
basic predicate 113
  equivalent term 689
BEGIN DECLARE SECTION statement 281
BETWEEN predicate 116
Binary
  data type 46
binary data string
  description 46
binary integer 43
bind 1
binding statement 1
bit data 44

© Copyright IBM Corp. 1982, 2002

BLOB data type
  in CREATE TABLE statement 358
BLOB function 148
built-in function 91
built-in type
  in CREATE TABLE statement 356
built-in-type
  in CREATE TABLE statement 356

C

C application program
  coding SQL statements 593
  host variable 85, 404
  INCLUDE SQLCA statement 527
  INCLUDE SQLDA statement 536
C NUL-terminated strings
  assignment 61
call level interface (CLI) 2
CALL statement 283, 481
CALLED ON NULL INPUT clause
  CREATE FUNCTION statement 319, 336
calling
  procedures 283
CASCADE delete rule
  description 5
  in ALTER TABLE statement 267, 276
  in CREATE TABLE statement 353, 363
CASE expression 106
CASE statement 482
CAST specification 109
casting
  between data types 54
  user-defined types 54
catalog 11
CCSID (coded character set identifier)
  conversion rules for assignments 61
  conversion rules for comparison 66
  conversion rules for string operations 71
  default 26
  description 26
  values 575
CDRA (character data representation architecture) 26
CEIL function 149
CEILING function 149
CHAR
  data type 44
  data type in CREATE TABLE statement 357
  function 150
character conversion
  character set 23
  code page 24
  code point 24
  coded character set 24
  encoding scheme 24
  substitution character 24
character data representation architecture (CDRA) 26
character data string
  assignment 60
  bit data 44
  comparison 65
character data string *(continued)*
  constant 73
  description 44
  empty 44
  mixed data 45
  SBCS data 45
  Unicode data 45
CHARACTER data type 357
character set 23
character-string constant 73
characters
  description 29
  digit 29
  letter 29
  special character 29
CHECK
  ALTER TABLE statement 277
CHECK clause
  ALTER TABLE statement 277
CHECK clause of CREATE TABLE statement 365
check constraint 6
CHECK OPTION clause of CREATE VIEW statement
  description 377
check-condition
  ALTER TABLE statement 273
  CREATE TABLE statement 361
CL_SCHED sample table 668
class-id
  description 33
CLOB data type
  in CREATE TABLE statement 357
CLOB function 155
CLOSE statement 287
closed state of cursor 431
COALESCE function 156
COBOL application program
  coding SQL statements 609
  host structure 89
  host variable 85, 404
  INCLUDE SQLCA statement 527
  INCLUDE SQLDA statement 538
code page 24
code point 24
coded character set 24
coding SQL statements
  in C applications 593
  in COBOL applications 609
  in Java applications 627
  in REXX applications 643
collection
  in SQL path 38
column
  description 3
  function
    See function
  grouping 244
  names 32, 79
  names in a result 236
  qualified names 79
  rules 248
COLUMN clause of COMMENT statement 290
column constraint of CREATE TABLE statement 360
column function 91
column function *(continued)*
  equivalent term 689
column in a group-by clause
  equivalent term 689
column-name
  description 32
  in ALTER TABLE statement 267, 271
  in AVG function 134
  in COMMENT statement 289, 291
  in COUNT function 135
  in CREATE INDEX statement 339
  in CREATE TABLE statement 353, 356
  in CREATE VIEW statement 376
  in DISTINCT operation of column functions 133
  in expressions 96
  in GRANT (Table or View Privileges) statement 418, 419
  in GROUP BY clause 244
  in HAVING clause 245
  in INSERT statement 423, 424
  in labeled-duration 96
  in LIKE predicate 120
  in MAX function 137
  in MIN function 138
  in NULL predicate 125
  in ORDER BY clause 251
  in search-condition of DELETE statement 387
  in select list 236
  in SUM function 140
  in UPDATE clause 254
  in UPDATE statement 466, 467
  in WHERE clause 243
  unqualified, length of 36
comment
  C 595
  COBOL 610
  in catalog tables 289
  REXX 646
  SQL 30, 266
  SQLJ 628
COMMENT statement
  column name qualification 79
  description 289
commit processing 12
COMMIT statement 294
comparison
  compatibility rules 58
  datetime values 66
  distinct type values 67
  numeric 64
  operation in SQL 58, 64
  predicate
    equivalent term 689
  predicate subquery
    equivalent term 689
  string 65
compatibility
  data types 58
  rules 58
composite key 3
compound statement 484
CONCAT function 157
CONCAT operator 96
concatenation operator 98

concurrency
  application 11
  with LOCK TABLE statement 428
condition handler
  declaring 488
condition-name
  description 32
CONNECT
  differences, type 1 and type 2 591
  statement, type 2 300
CONNECT statement 296
connected state 22
connecting to a data source
  SQLJ 629
connection
  SQL 20
connection state 459
  CONNECT (Type 2) statement 20
connection states
  application process 22
  distributed unit of work 20
  remote unit of work 18
constant
  character string 73
  decimal 73
  floating-point 73
  graphic string 74
  in expressions 96
  in IN predicate 118
  in labeled-duration 96
  integer 73
  SQL 73
CONSTRAINT clause of ALTER TABLE statement 272
CONSTRAINT clause of CREATE TABLE statement 360, 364
constraint-name
  description 32
  in ALTER TABLE statement 267
  in CREATE TABLE statement 353
  length 36
constraint, check 6
CONTINUE clause of WHENEVER statement 475
control statements 477
conversion of numbers
  precision 59
  scale 59
conversion rules for assignments 61
conversion rules for comparisons 61
conversion rules for operations that combine strings 71
correlated reference 83, 243
  equivalent term 689
correlation name
  defining 79
  qualifying a column name 79
correlation-name
  description 32
  in DELETE statement 386, 387
  in SELECT clause 235
  in UPDATE statement 466, 467
  length 36
COS function 158
COUNT function 135
COUNT_BIG function 136
CREATE ALIAS statement 8, 303
CREATE DISTINCT TYPE statement 304, 309
CREATE FUNCTION (external scalar) statement 324
CREATE FUNCTION (sourced) statement 331
CREATE FUNCTION (SQL) statement 337
CREATE FUNCTION statement 310, 314, 325, 332
CREATE INDEX statement 338
CREATE PROCEDURE (External) statement 341
CREATE PROCEDURE (SQL) statement 348
CREATE PROCEDURE statement 340
  assignment statement 480
  CALL statement 481
  CASE statement 482
  compound statement 484
  condition handlers 488
  DECLARE statement 484
  GET DIAGNOSTICS statement 491
  GOTO statement 493
  handler statement 488
  IF statement 494
  LEAVE statement 496
  LOOP statement 497
  REPEAT statement 498
  RESIGNAL statement 500
  RETURN statement 502
  SIGNAL statement 504
  SQL control statement 477
  variables 484
  WHILE statement 506
CREATE TABLE statement 353
CREATE TRIGGER statement 368
CREATE VIEW statement 8, 376
CS (cursor stability) isolation level 15
CURRENT
  clause of RELEASE statement 440
current connection state 21
CURRENT DATE special register 76
CURRENT PATH special register 76
  SET PATH statement 461
current server
  designating
    CONNECT (Type 2) statement 300
CURRENT SERVER special register 77
CURRENT TIME special register 77
CURRENT TIMESTAMP special register 77
CURRENT TIMEZONE special register 78
cursor
  closed by error
    FETCH statement 407
    UPDATE statement 469
  closed state 431
  closing 287
    CONNECT (Type 2) statement 300
  current row 407
  defining 381
  moving position 406
  name 32
cursor *(continued)*
  positions for open 407
  preparing 429
  read-only status, conditions for 382
  updatability, determining 382
cursor stability 15
cursor-name
  description 32
  in CLOSE statement 287
  in DECLARE CURSOR statement 381
  in DELETE statement 386, 388
  in FETCH statement 406
  in OPEN statement 429
  in UPDATE statement 466, 469
  length 36

D

data access classification 518
data representation considerations 23
data type
  binary string 46
  character string 44
  datetime values 49
  description 42
  distinct 51, 304
  function 314, 325, 332
  graphic string 45
  in CREATE TABLE statement 356
  in SQLCA 525
  in SQLDA 530
  numbers 43
  result columns 237
  user-defined 51
data types
  casting between 54
  equivalent Java and SQL 640
  promotion 53
data-type
  in ALTER TABLE statement 267, 271
  in CAST specification 110
  in CREATE TABLE statement 353, 356
database manager limits 511, 512, 513
date
  duration 101
  strings 50
DATE
  arithmetic operations 102
  assignment 62
  CREATE TABLE statement 359
  data type 49, 359
  function 159
date and time
  format 151
datetime
  arithmetic operations 101
  assignment 62
  comparisons 66
  data types
    default date format 49
    default time format 49
    description 49
    string representation 49
  format
    EUR 50, 150

datetime (continued)
format (continued)
ISO 50, 150
JIS 50, 150
USA 50, 150
limits 511
Datetime Host Variables 49
DAY function 161
DAY labeled duration 96, 100
DAYOFWEEK function 162
DAYOFWEEK_ISO function 163
DAYOFYEAR function 164
DAYS function 165
DAYS labeled duration 96, 100
DBCLOB
function 166
DBCLOB data type
in CREATE TABLE statement 358
DBCS (double-byte character set) data
description 46
strings 45
within mixed data 45
DBINFO
clause of CREATE FUNCTION statement 319
DEC data type 353, 356
decimal
arithmetic in SQL 97
constant 73
data type 43
numbers 43
DECIMAL
data type 353
DECIMAL function 167
decimal point 75
declarations in a program 421
DECLARE CURSOR statement 381
DECLARE statement 484
DEFAULT clause
in ALTER TABLE statement 271
in CREATE TABLE statement 359
default date format 49
default decimal point 75
default decimal separator character
description 44
DEFAULT keyword
in INSERT statement 423, 424
default solation level 13
default time format 49
degree
of table
equivalent term 689
DEGREES function 169
deletable
view 379
DELETE
clause of GRANT (Table or View Privileges) statement 419
clause of REVOKE (Table and View Privileges) statement 453
in ON DELETE clause of ALTER TABLE statement 276
in ON DELETE clause of CREATE TABLE statement 363
statement 386
delete rule for referential constraint 5
delete-connected table 5
deleting SQL objects 395
delimited identifier in SQL 31
delimiter token 30
DEPARTMENT sample table 668
dependent privilege 453
dependent row 4
dependent table 4
DESC clause
CREATE INDEX statement 339
select-statement 252
descendent row 4
descendent table 4
DESCRIBE statement 391
descriptor-name
description 32
in C 593
in COBOL 610
in DESCRIBE statement 391
in EXECUTE statement 401
in FETCH statement 406
in OPEN statement 429, 430
in PREPARE statement 433
in REXX 644
DETERMINISTIC clause
CREATE FUNCTION statement 318, 334
digit 29
DIGITS function 170
dirty read 16
DISALLOW PARALLEL clause
CREATE FUNCTION statement 321
DISTINCT
clause of subselect 235
COUNT_BIG function 136
keyword
AVG function 134
column function 133
COUNT function 135
MAX function 137
MIN function 138
SUM function 140
STDDEV function 139
VAR function 141
VARIANCE function 141
distinct type
assignment 63
comparisons 67
CREATE DISTINCT TYPE statement 304
description 51
DISTINCT TYPE clause
COMMENT statement 291
REVOKE (Distinct Type Privileges) statement 444
distinct type name
in CREATE TABLE statement 359
in GRANT (Distinct Type Privileges) statement 410
distinct-type-name
description 32
in COMMENT statement 289
in CREATE TABLE statement 359
in DROP statement 396
in REVOKE (Distinct Type Privileges) statement 444
distributed data
CONNECT statement 17
distributed data (continued)
RELEASE statement 440
SET CONNECTION statement 459
distributed relational database
application requester 16
application server 16
considerations for using 520
data representation considerations 23
remote unit of work 18
use of extensions to IBM SQL on unlike application servers 520
distributed relational database architecture (DRDA) 16
distributed unit of work
mixed environment 516
division by zero 107
dormant connection state 21
DOUBLE
function 171
DOUBLE PRECISION data type in CREATE TABLE statement 357
DOUBLE_PRECISION function 171
double-precision floating-point numbers 43
DRDA (Distributed Relational Database Architecture) 16
driver, JDBC
registering with DriverManager 627
DROP CHECK clause of ALTER TABLE statement 278
DROP FOREIGN KEY clause of ALTER TABLE statement 278
DROP PRIMARY KEY clause of ALTER TABLE statement 278
DROP statement 395
DROP UNIQUE clause of ALTER TABLE statement 278
duplicate rows in fullselect 248
duration
date 101
labeled 100
time 101
timestamp 101
dynamic select 264
dynamic SQL
description 2
EXECUTE IMMEDIATE statement 404
EXECUTE statement 401
execution 263
obtaining information with DESCRIBE 391
preparation 263
PREPARE statement 433
SQLDA 529
statements allowed 516
use of SQL path 38

E

embedded SQL 262
Embedded SQL for Java (SQLJ) 2
EMP_PHOTO sample table 668
EMP_RESUME sample table 669
EMPLOYEE sample table 669
EMPPROJACT sample table 672
empty character string 44

encoding scheme 24
END DECLARE SECTION statement 400
ending a unit of work 294, 455
equivalent terms 689
error
  closes cursor 431
  DELETE statement 388
  FETCH statement 407
  return code 264, 539
  UPDATE statement 469
error handling
  SQLJ 639
escape character 31
ESCAPE clause of LIKE predicate 122
EUR (IBM European standard)
  argument in CHAR function 150
evaluation order 105
exclusive locks 14
EXCLUSIVE option of LOCK TABLE statement 428
EXECSQL REXX command 643, 646
executable statement 262, 263
EXECUTE
  in GRANT (Function or Procedure Privileges) statement 413
  in GRANT (package privileges) statement 416
  in REVOKE (Function and Procedure Privileges) statement 447
  in REVOKE (package privileges) statement 450
EXECUTE IMMEDIATE statement 404
EXECUTE privilege 412, 416, 447, 450
EXECUTE statement 401
EXISTS predicate 117
EXP function 173
exposed name 80
expression 336
  arithmetic operators 96
  CASE expression 106
  CAST specification 109
  concatenation operator 98
  datetime operands 100
  decimal arithmetic in SQL 97
  decimal operands 97
  description 96, 129
  distinct type operands 98
  floating-point operands 98
  in ABS function 143
  in ACOS function 144
  in ASIN function 145
  in ATAN function 146
  in ATAN2 function 147
  in AVG function 134
  in basic predicate 113
  in BETWEEN predicate 116
  in BLOB function 148
  in CEILING function 149
  in CHAR function 150
  in CLOB function 155
  in COALESCE function 156
  in CONCAT function 157
  in COS function 158
  in COUNT function 135
  in COUNT_BIG function 136
  in DATE function 159
expression *(continued)*
  in DAY function 161
  in DAYOFWEEK function 162
  in DAYOFWEEK_ISO function 163
  in DAYOFYEAR function 164
  in DAYS function 165
  in DBCLOB function 166
  in DECIMAL function 167
  in DEGREES function 169
  in DIGITS function 170
  in DOUBLE function 171
  in DOUBLE_PRECISION function 171
  in EXP function 173
  in FLOAT function 174
  in FLOOR function 175
  in GRAPHIC function 176
  in HEX function 177
  in HOUR function 178
  in IN predicate 118
  in INSERT statement 423, 424
  in INTEGER function 179
  in JULIAN_DAY function 180
  in labeled-duration 96
  in LCASE function 181
  in LEFT function 182
  in LENGTH function 183
  in LN function 184
  in LOCATE function 185
  in LOG10 function 187
  in LOWER function 188
  in LTRIM function 189
  in MAX function 137
  in MICROSECOND function 190
  in MIDNIGHT_SECONDS function 191
  in MIN function 138
  in MINUTE function 192
  in MOD function 193
  in MONTH function 194
  in NULLIF function 195
  in POSSTR function 196
  in POWER function 198
  in quantified predicate 114
  in QUARTER function 199
  in RADIANS function 200
  in RAND function 201
  in REAL function 202
  in ROUND function 203
  in RTRIM function 205
  in scalar functions 142
  in SECOND function 206
  in SELECT clause 235
  in SIGN function 207
  in SIN function 208
  in SMALLINT function 209
  in SPACE function 210
  in SQRT function 211
  in STDDEV function 139
  in subselect 235
  in SUBSTR function 212
  in SUM function 140
  in TAN function 214
  in TIME function 215
  in TIMESTAMP function 216
  in TRANSLATE function 218
  in TRUNC function 220
expression *(continued)*
  in TRUNCATE function 220
  in UCASE function 222
  in UPDATE statement 466, 468
  in UPPER function 223
  in VALUE function 224
  in VALUES statement 472
  in VAR function 141
  in VARCHAR function 225
  in VARGRAPHIC function 227
  in VARIANCE function 141
  in WEEK function 230
  in WEEK_ISO function 231
  in YEAR function 232
  integer operands 97
  precedence of operation 105
  two decimal operands 97
  two integer operands 97
  without operators 96
EXTERNAL ACTION clause
  CREATE FUNCTION statement 319, 335
EXTERNAL clause
  CREATE FUNCTION statement 322
external function program
  call-type parameter 655
  dbinfo parameter 656
  diagnostic-message parameter 655
  qualified-function-name parameter 655
  scratchpad parameter 655
  specific-name parameter 655
  SQL-argument parameter 653
  SQL-argument-ind parameter 654
  SQL-result parameter 653
  SQL-result-ind parameter 654
  SQL-state parameter 654
external procedure program
  dbinfo parameter 659
  diagnostic-message parameter 659
  qualified-procedure-name parameter 659
  specific-name parameter 659
  SQL-argument parameter 657
  SQL-argument-ind parameter 658
  SQL-argument-ind-array parameter 658
  SQL-state parameter 659
external-program-name
  description 33

F

FENCED
  clause of CREATE FUNCTION statement 320
FETCH statement 406
fixed-length string 44, 45
FLOAT data type in CREATE TABLE statement 356, 357
FLOAT function 174
floating-point
  constant 73
  numbers 43
FLOOR function 175
FOR BIT DATA clause of CREATE TABLE statement 357

FOR UPDATE OF clause
  prohibited in views 379
foreign key 4
FOREIGN KEY clause
  ALTER TABLE statement 275
  of ALTER TABLE statement 275
  of CREATE TABLE statement 362
FREE LOCATOR statement 409
FROM clause
  correlation clause 239
  DELETE statement 387
  joined-table 241
  nested table expression 239
  of subselect 239
  PREPARE statement 434
  REVOKE (Distinct Type Privileges) statement 444
  REVOKE (Function and Procedure Privileges) statement 449
  REVOKE (package privileges) statement 450
  REVOKE (Table and View Privileges) statement 453
  SELECT INTO statement 457
  table reference 239
fullselect
  conversion rules for operations that combine strings 71
  description 233, 248
  equivalent term 689
  examples of 249
  in INSERT statement 423, 425
  ORDER BY clause 251
  subselect component 234
  UPDATE clause 254
function 9
  best fit 93
  built-in 91
  column 91
    AVG 134
    COUNT 135
    COUNT_BIG 136
    description 133
    MAX 137
    MIN 138
    STDDEV 139
    SUM 140
    VARIANCE or VAR 141
  description 129
  dropping 397, 398
  external 91
  function 310
  in expressions 96
  in labeled-duration 96
  invocation 95
  nesting 142
  resolution 92
  scalar 91
    ABS 143
    ACOS 144
    ASIN 145
    ATAN 146
    ATAN2 147
    BLOB 148
    CEIL 149
    CEILING 149
    CHAR 150
function *(continued)*
  scalar *(continued)*
    CLOB 155
    COALESCE 156
    CONCAT 157
    COS 158
    DATE 159
    DAY 161
    DAYOFWEEK 162
    DAYOFWEEK_ISO 163
    DAYOFYEAR 164
    DAYS 165
    DBCLOB 166
    DECIMAL 167
    DEGREES 169
    description 142
    DIGITS 170
    DOUBLE 171
    DOUBLE_PRECISION 171
    EXP 173
    FLOAT 174
    FLOOR 175
    GRAPHIC 176
    HEX 177
    HOUR 178
    INTEGER 179
    JULIAN_DAY 180
    LCASE 181
    LEFT 182
    LENGTH 183
    LN 184
    LOCATE 185
    LOG10 187
    LOWER 188
    LTRIM 189
    MICROSECOND 190
    MIDNIGHT_SECONDS 191
    MINUTE 192
    MOD 193
    MONTH 194
    NULLIF 195
    POSSTR 196
    POWER 198
    QUARTER 199
    RADIANS 200
    RAND 201
    REAL 202
    ROUND 203
    RTRIM 205
    SECOND 206
    SIGN 207
    SIN 208
    SMALLINT 209
    SPACE 210
    SQRT 211
    SUBSTR 212
    TAN 214
    TIME 215
    TIMESTAMP 216
    TRANSLATE 218
    TRUNCATE 220
    UCASE 222
    UPPER 223
    VALUE 224
    VALUE (see COALESCE) 156
    VARCHAR 225
    VARGRAPHIC 227
function *(continued)*
  scalar *(continued)*
    WEEK 230
    WEEK_ISO 231
    YEAR 232
  sourced 91
  SQL 91
  types 91
  user-defined 91
FUNCTION clause
  COMMENT statement 291
  DROP statement 396
  GRANT (Function or Procedure Privileges) statement 413
  REVOKE (Function and Procedure Privileges) statement 447
  REVOKE (Function or Procedure Privileges) statement 447
function invocation
  syntax 92
function path 52
function reference
  syntax 92
function resolution 38
function-name
  function 34
  in COMMENT statement 289
  in DROP statement 396
  in GRANT (Function or Procedure Privileges) statement 412
  in REVOKE (Function and Procedure Privileges) statement 447
functions
  attributes of arguments 660
  coding programs for external functions 653
  DBINFO 662
  description 91
  parameter passing to C or COBOL 653
  parameter passing to Java 656
  scratch pad 666
  SCRATCHPAD 666

G

GET DIAGNOSTICS statement 491
GO TO clause of WHENEVER statement 475
GOTO statement 493
GRANT (Distinct Type Privileges) statement 410
GRANT (Function or Procedure Privileges) statement 412
GRANT (package privileges) statement 416
GRANT (Table or View Privileges) statement 418
GRAPHIC
  function 176
graphic data string
  DBCS data 46
  Unicode data 46
GRAPHIC data type
  in ALTER TABLE statement 267
  in CREATE TABLE statement 358

graphic string
constant 74
definition 45
GROUP BY clause
cannot join view 379
SELECT INTO statement 457
subselect
intermediate results 244
results 236
grouping column 244

H

handlers
declaring 488
Handling SQL Errors and Warnings
COBOL 612
HAVING clause
results with subselect 236
SELECT INTO statement 457
subselect 245
held connection state 21
HEX function 177
host label
WHENEVER statement 475
host structure 89
C 603
COBOL 620
host variable
C 597
COBOL 612
description 34, 85
EXECUTE IMMEDIATE statement 404
FETCH statement 406
indicator variable 86
LOB locator 88
naming a structure
C program 603
references to 85
REXX 648
SELECT INTO statement 457
substitution for parameter markers 401
host variable followed by an indicator variable
equivalent term 689
host-identifier
description 86
length 36
host-identifiers
description 31
host-label
description 34
host-variable
description 34, 85, 86
in Java 87
general use in SQL statements 86
in CONNECT statement 296
in EXECUTE IMMEDIATE statement 404
in EXECUTE statement 401
in expressions 96
in FETCH statement 406
in IN predicate 118
in labeled-duration 96
in LIKE predicate 120, 122
host-variable *(continued)*
in OPEN statement 429
in PREPARE statement 433, 434
in SELECT INTO statement 457
HOUR function 178
HOUR labeled duration 96, 100
HOURS labeled duration 96, 100

I

identifiers
host identifier 31
limits 37, 509, 510
naming conventions 31
SQL
delimited 31
description 31
limits 36
ordinary 31
IF statement 494
IMMEDIATE keyword of EXECUTE IMMEDIATE statement 404
IN EXCLUSIVE MODE clause of LOCK TABLE statement 428
IN predicate 118
IN SHARE MODE clause of LOCK TABLE statement 428
IN_TRAY sample table 674
INCLUDE statement 421
index
dropping 398
INDEX clause
COMMENT statement 292
INDEX keyword
CREATE INDEX statement 338
DROP statement 398
GRANT (Table or View Privileges) statement 419
REVOKE (Table and View Privileges) statement 453
index-name
description 34
in COMMENT statement 289
in CREATE INDEX statement 338
in DROP statement 395, 398
unqualified, length of 36
index, definition of 8
indicator array 89
INDICATOR keyword 86
indicator variable
C 603
COBOL 619
in EXECUTE IMMEDIATE statement 404
REXX 649
infix operators 97
INNER JOIN clause
in FROM clause 242
INSERT
clause of GRANT (Table or View Privileges) statement 419
clause of REVOKE (Table and View Privileges) statement 453
statement 423
insert rule with referential constraint 5
insert rules with INSERT statement 425
insertable
view 379
INT data type 356
integer
constant 73
in ALTER TABLE statement 267
in C VARGRAPHIC structured form 601
in CREATE TABLE statement 353, 357
in ORDER BY clause 251, 252
INTEGER
data type 43, 356
INTEGER function 179
interactive entry of SQL statements 264
interactive SQL 2
INTO clause
in PREPARE statement 433
INTO keyword
DESCRIBE statement 391
FETCH statement 406
in INSERT statement 424
SELECT INTO statement 457
VALUES INTO statement 473
invoking SQL statements 262
IS clause of COMMENT statement 292
ISO (International Standards Organization)
argument in CHAR function 150
isolation clause
select-statement 257
isolation level
comparison 15
cursor stability 15
default 13
description 13
read stability 14
repeatable read 14
uncommitted read 15
isolation levels
in DELETE statement 257
isolation-clause
DELETE statement 388
INSERT statement 425
SELECT INTO statement 457
UPDATE statement 469
iterator
for positioned DELETE 637
for positioned UPDATE 637

J

jar-name
description 33
Java
equivalent SQL data types 640
Java application program
coding SQL statements 627
Java Database Connectivity (JDBC) 2
JIS (Japanese Industrial Standard)
argument in CHAR function 150
JOIN clause
in FROM clause 242
JULIAN_DAY function 180

K

key
  ALTER TABLE statement 274
  composite 3
  CREATE TABLE statement 361
  foreign 4
  parent 4
  primary 3
  primary index 3
  unique 3
  unique index 3

L

label
  description 34
label, GOTO 493
labeled duration 100
labeled-duration 96
LANGUAGE
  clause of CREATE FUNCTION statement 317, 334
large integers 43
large object (LOB)
  locator variable 88
large object location, definition 47
LCASE function 181
LEAVE statement 496
LEFT function 182
LEFT JOIN clause
  in FROM clause 242
LEFT OUTER JOIN clause
  in FROM clause 242
length attribute of column 44, 45
LENGTH function 183
letter 29
LIKE predicate 120
limits
  database manager 511, 512, 513
  datetime 511
  DB2 UDB SQL 509
  identifier 37, 509, 510
  numeric 510
  string 510, 511
literal
  constant
    equivalent term 689
literals 73
LN function 184
LOB
  locator variable 88
  locator, definition 47
LOB Locators
  assignment 64
LOCATE function 185
locator
  declaring host variable 88
  definition 47
  FREE LOCATOR statement 409
LOCK TABLE statement 428
locks
  description 11
  exclusive 14
  LOCK TABLE statement 428
  share 14
  UPDATE statement 469
LOG10 function 187
logical operator 126
long string
  limitations 44, 135, 136, 137, 138, 159, 161, 162, 163, 164, 165, 167, 171, 176, 177, 178, 179, 180, 188, 190, 191, 192, 194, 199, 206, 209, 215, 216, 218, 223, 225, 227, 230, 231, 232
LOOP statement 497
LOWER function 188
LTRIM function 189

M

MAX function 137
method-id
  description 33
MICROSECOND function 190
MICROSECOND labeled duration 96, 100
MICROSECONDS labeled duration 96, 100
MIDNIGHT_SECONDS function 191
MIN function 138
MINUTE function 192
MINUTE labeled duration 96, 100
MINUTES labeled duration 96, 100
mixed data
  description 45
  EBCDIC
    shift-in character xii
    shift-out character xii
  string assignment 61
mixed string
  truncated during assignment 61
mixed strings
  assignment 61
MOD function 193
MODE keyword of LOCK TABLE statement 428
MONTH function 194
MONTH labeled duration 96, 100
MONTHS labeled duration 96, 100

N

NAME clause
  CREATE FUNCTION statement 322
name qualification 38
named iterator
  example 636
  renaming result table columns for 637
  SQLJ iterator 636
naming conventions
  C 596
  COBOL 611
  REXX 647
  SQL 32
NO DBINFO clause
  CREATE FUNCTION statement 319
NO EXTERNAL ACTION clause
  CREATE FUNCTION statement 319, 335
NO SCRATCHPAD clause
  CREATE FUNCTION statement 321
nonexecutable statement 262, 263
nonexposed name 80
nonrepeatable read 16
NOT
  in BETWEEN predicate 116
  in IN predicate 118
  in LIKE predicate 120
  in NULL predicate 125
  operator in search conditions 126
NOT DETERMINISTIC clause
  CREATE FUNCTION statement 318, 334
NOT FOUND clause of WHENEVER statement 475
NOT NULL clause of CREATE TABLE statement 353, 360
NOT NULL PRIMARY KEY clause of CREATE TABLE statement 353
NOT NULL UNIQUE clause of CREATE TABLE statement 353
NUL in C 596
NUL terminator 599
NUL-terminated host variable 598, 600
NULL
  in CAST specification 110
  in VALUES statement 472
  keyword in UPDATE statement 467
  keyword SET NULL delete rule
    description 5
    in ALTER TABLE statement 276
    in CREATE TABLE statement 363
  predicate 125
  value in C 596
NULL keyword
  in INSERT statement 423, 424, 425
null string in REXX 647
null value, SQL
  assigned to host variable 457
  assignment 59
  contrasted with null value in C 596
  contrasted with null value in REXX 647
  definition 42
  duplicate rows 235
  grouping columns 244
  result columns 236
  specified by indicator variable 86
NULLIF function 195
number of items in a select list
  equivalent term 690
numbers
  default decimal separator character 44
  precision 43
  scale 43
  SQL 43
  truncation of 59
numeric
  assignment 59
  comparisons 64
  data type 43
  data types
    default decimal separator character 44
    string representation 44
  limits 510

NUMERIC
  data type 356

O

object table 82
ON clause
  CREATE INDEX statement 338
  GRANT (package privileges) statement 416
  GRANT (Table or View Privileges) statement 419
  REVOKE (package privileges) statement 450
  REVOKE (Table and View Privileges) statement 453
ON DISTINCT TYPE clause
  REVOKE (Distinct Type Privileges) statement 444
ON TYPE clause
  GRANT (Distinct Type Privileges) statement 410
open state of cursor 407
OPEN statement 429
operand
  datetime 100
  decimal 97
  distinct type 98
  floating-point 98
  integer 97
  string 98
operands of in list
  result data type 68
operation
  assignment 58
  comparison 64
  description 58
  precedence 105
operator
  arithmetic 96
  concatenation 98
  logical 126
  string 98
OPTIMIZE FOR clause
  select-statement 256
OR
  operator in search condition 126
  truth table 126
ORDER BY clause
  prohibited in views 379
  select-statement 251
order of evaluation of operators 105
ordinary identifier in SQL 31
ordinary token 30
ORG sample table 674
outer join
  See also LEFT OUTER JOIN clause
  See RIGHT OUTER JOIN clause
outer reference
  equivalent term 689
ownership 28

P

package
  dropping 398
package and access plan 9
PACKAGE clause
  DROP statement 398
  GRANT (Package Privileges) statement 416
  REVOKE (Package Privileges) statement 450
package-name
  description 34
  in DROP statement 395, 398
  in GRANT (package privileges) statement 416
  in REVOKE (package privileges) statement 450
  unqualified, length of 36
padding on string assignment 61
parameter marker
  EXECUTE statement 401, 429
  in PREPARE statement 434
  OPEN statement 430
  replacement 402, 431
  rules 434
  typed 434
  untyped 434
  usage in expressions, predicates and functions 434
parameter style
  DB2SQL 657
  GENERAL 657
  GENERAL WITH NULLS 657
PARAMETER STYLE clause
  CREATE FUNCTION statement 318
parameter-marker
  in CAST specification 110
parameter-name
  description 34
parent key 4
parent row 4
parent table 4
parentheses used to change order of evaluation of
  expression 105
  UNION operation 248
path
  function resolution 93
performance
  isolation clause 257
  OPTIMIZE FOR clause 256
phantom row 14, 16
Positioned DELETE statement 386
positioned iterator
  example 635
  SQLJ iterator 635
Positioned UPDATE statement 466
POSSTR function 196
POWER function 198
precedence
  level 105
  operation 105
precision of numbers
  assignment 59
  comparisons 64
  description 43
  determined by SQLLEN variable 533
  results of arithmetic operations 97
predicate
  basic 113
predicate *(continued)*
  BETWEEN 116
  description 112
  EXISTS 117
  IN 118
  in search condition 126
  LIKE 120
  NULL 125
  quantified 114
prefix operator 97
PREPARE statement 433
prepared SQL statement
  dynamically prepared by PREPARE 433, 438
  executing 401
  obtaining information
    by INTO with PREPARE 392
  obtaining information with DESCRIBE 391
  SQLDA provides information 529
  statements allowed 516
preparing statements 1
primary index 3
primary key 3
PRIMARY KEY clause
  ALTER TABLE statement 274
  CREATE TABLE statement 361
PRIMARY KEY clause of ALTER TABLE statement 272
PRIMARY KEY clause of CREATE TABLE statement 360
privileges
  description 27
  granting 410, 412, 416, 418
  revoking 444, 446, 450, 452
procedure 9
  authorization for creating 341, 348
  creating, SQL statement instructions 340, 341, 348
  dropping 398
PROCEDURE clause
  DROP statement 398
procedure-name
  in COMMENT statement 289
  in DROP statement 398
  in GRANT (Function or Procedure Privileges) statement 412
  in REVOKE (Function and Procedure Privileges) statement 447
  procedure 34
PROCEDURE
  clause of COMMENT statement 292
procedures
  attributes of arguments 660
  coding programs for external procedures 653
  DBINFO 662
  parameter passing 657
  parameter passing to Java 660
program preparation 1
PROGRAM synonym for PACKAGE
  DROP statement 398
  GRANT (Package Privileges) statement 416
PROJACT sample table 675
PROJECT sample table 676

promoting
  data types 53
  precedence 53
PUBLIC clause
  GRANT (Distinct Type Privileges) statement 411
  GRANT (Function or Procedure Privileges) statement 415
  GRANT (package privileges) statement 417
  GRANT (Table or View Privileges) statement 420
  REVOKE (Distinct Type Privileges) statement 444
  REVOKE (Function and Procedure Privileges) statement 449
  REVOKE (package privileges) statement 450
  REVOKE (Table and View Privileges) statement 453
publications, related xiii

Q

qualified column names 79
qualifier
  reserved 691
quantified predicate 114
QUARTER function 199
queries 233
query
  expression
    equivalent term 689
  specification
    equivalent term 689
question mark
  EXECUTE statement 401, 429
  PREPARE statement 434

R

RADIANS function 200
RAND function 201
READ ONLY clause
  select-statement 255
read stability 14
read-only
  READ ONLY clause 255
  view 379
REAL data type in CREATE TABLE statement 357
REAL function 202
recovery of applications 11
REFERENCES clause
  ALTER TABLE statement 267, 273, 275
  CREATE TABLE statement 353, 361
  FOREIGN KEY clause 273, 361
  GRANT (Table or View Privileges) statement 419
  REVOKE (Table and View Privileges) statement 453
references to host variables 85
referential constraint 4
referential cycle 4
referential integrity 4
relational database 1
RELEASE statement 440
release-pending connection state 21
remote unit of work 18
RENAME statement 442
REPEAT statement 498
repeatable read 14
reserved
  qualifiers 691
  schema names 691
  words 691
RESET
  clause of CONNECT statement 301
RESIGNAL statement 500
RESTRICT delete rule
  description 5
  in ALTER TABLE statement 267, 276
  in CREATE TABLE statement 353, 363
result
  equivalent term 690
result columns of subselect 237
result data type
  arguments of COALESCE 68
  operands 68
  result expressions of CASE 68
  UNION 68
result expressions of CASE
  result data type 68
result sets
  returning from a SQL procedure 487
result specification
  equivalent term 689
result table 3
result table created by a group-by or having clause
  equivalent term 690
result view created by a group-by or having clause
  equivalent term 690
RESULT_STATUS
  GET DIAGNOSTICS statement 491
result-expression
  in CASE specification 106
retrieval
  assignment 61
retrieving rows in SQLJ
  named iterator example 636
  positioned iterator example 635
  with named iterators 636
return code 264, 539
RETURN statement 336, 502
returning result sets 487
RETURNS clause of CREATE FUNCTION statement 317, 327, 334
RETURNS NULL ON NULL INPUT clause
  CREATE FUNCTION statement 319
REVOKE (Distinct Type Privileges) statement 444
REVOKE (Function or Procedure Privileges) statement 446
REVOKE (package privileges) statement 450
REVOKE (Table and View Privileges) statement 452
REXX application program
  coding SQL statements 643
  host variable 85
RIGHT JOIN clause
  in FROM clause 242
RIGHT OUTER JOIN clause
  in FROM clause 242
rollback description 12
ROLLBACK statement 455
ROUND function 203
routine 9
routines
  attributes of arguments 660
  coding program for external routines 653
  DBINFO 662
  parameter passing 653
  parameter passing for functions written in C or COBOL 653
  parameter passing for functions written in Java 656
  parameter passing for procedures 657
  parameter passing for procedures written in Java 660
  scratch pad 666
row
  deleting 386
  dependent 4
  descendent 4
  description 3
  inserting 423
  parent 4
  self-referencing 4
ROW_COUNT
  GET DIAGNOSTICS statement 491
RR (repeatable read) isolation level 14
RS (read stability) isolation level 14
RTRIM function 205
RUN privilege 416
run-time authorization ID 41

S

SALES sample table 678
sample tables 667
SBCS (single-byte character set) data
  description 45
  within mixed data 45
scalar function 91
scale of data
  determined by SQLLEN variable 532
scale of numbers
  assignment 59
  comparisons 64
  description 43
  determined by SQLLEN variable 533
  results of arithmetic operations 97
schema 34
  system 3
schema-name
  description 34
  length 36
  reserved names 691
SCRATCHPAD clause
  CREATE FUNCTION statement 321

search condition
DELETE statement 387
description 126
HAVING clause 245
in JOIN clause 241
order of evaluation 126
WHERE clause 243
search-condition
description 126
in CASE specification 107
in DELETE statement 386, 387
in HAVING clause 245
in UPDATE statement 466, 469
Searched DELETE statement 386
Searched UPDATE statement 466
searched-when-clause
in CASE specification 106
SECOND function 206
SECOND labeled duration 96, 100
SECONDS labeled duration 96, 100
SELECT
clause of GRANT (Table or View Privileges) statement 419
clause of REVOKE (Table and View Privileges) statement 453
clause of subselect 234
select-statement 250
SELECT INTO statement 457
select list
application 236
description 235
notation 235
SELECT statement 456
select-statement
description 233, 250
examples of 258
in DECLARE CURSOR statement 381, 382
in OPEN statement 429
in UPDATE clause 254
self-referencing row 4
self-referencing table 4
separator
comment 30
space 30
server-name
description 34
in CONNECT statement 296
length 36
SET clause of UPDATE statement 467
SET CONNECTION statement 459
set function
equivalent term 689
SET NULL delete rule
description 5
in ALTER TABLE statement 276
in CREATE TABLE statement 363
SET PATH statement
detailed description 461
SET statement 480
SET transition-variable statement 464
share locks 14
SHARE option of LOCK TABLE statement 428
shift-in character xii, 100
shift-out character xii, 100
SIGN function 207
SIGNAL ON ERROR in REXX 647
SIGNAL statement 504
simple-when-clause
in CASE specification 106
SIN function 208
single-precision floating-point numbers 43
single-row select 457
small integers 43
SMALLINT data type 356
SMALLINT function 209
SOME quantified predicate 114
SOURCE clause of CREATE FUNCTION statement 328
space 29
SPACE function 210
special character 29
special register
CURRENT DATE 76, 77
CURRENT PATH 76
CURRENT SERVER 77
CURRENT TIME 77
CURRENT TIMESTAMP 77
CURRENT TIMEZONE 78
description 76
USER 78
special-register
in expressions 96
in IN predicate 118
SPECIFIC clause
CREATE FUNCTION statement 318, 327, 334
DROP statement 398
GRANT (Function or Procedure Privileges) statement 414
REVOKE (Function and Procedure Privileges) statement 448
specific-name
description 35
in DROP statement 398
in GRANT (Function or Procedure Privileges) statement 414
in REVOKE (Function and Procedure Privileges) statement 448
SQL
equivalent Java data types 640
SQL (Structured Query Language) 1
call level interface (CLI) 2
dynamic
statements allowed 516
Embedded SQL for Java (SQLJ) 2
Java Database Connectivity (JDBC) 2
Open Database Connectivity (ODBC) 2
SQL 1999 Core standard ix
SQL comments 266
SQL Control statement
SQL procedure statement 479
SQL control statements 477
SQL data acess clause
CREATE FUNCTION statement 335
SQL limits 509
SQL path 38, 52
function resolution 93
SQL procedure
assignment statement 480
CALL statement 481
SQL procedure *(continued)*
CASE statement 482
compound statement 484
condition handler statement 488
condition handlers 488
DECLARE statement 484
GET DIAGNOSTICS statement 491
GOTO statement 493
IF statement 494
LEAVE statement 496
LOOP statement 497
REPEAT statement 498
RESIGNAL statement 500
RETURN statement 502
SET statement 480
SIGNAL statement 504
variables 484
WHILE statement 506
SQL return code 264, 539
SQL statement
CREATE ALIAS 303
CREATE DISTINCT TYPE 304, 309
CREATE FUNCTION 310, 314, 325, 332
CREATE FUNCTION (external scalar) 324
CREATE FUNCTION (sourced) 331
CREATE FUNCTION (SQL) 337
CREATE PROCEDURE 340
CREATE PROCEDURE (External) 341
CREATE PROCEDURE (SQL) 348
format in SQLJ 627
FREE LOCATOR 409
handling errors in SQLJ 639
ROLLBACK 455
SQL statements
CALL 283
characteristics 515
CONNECT (Type 2) 300
CONNECT differences 591
CREATE TRIGGER 368
data access classification 518
RELEASE 440
RENAME 442
SET CONNECTION 459
SET PATH 461
SET transition-variable 464
SQL variables 484
SQL-label
description 35
SQL-parameter-name
description 35
SQL-variable-name
description 35
SQLCA (SQL communication area)
C 593
COBOL 609
contents 525
description 525
entry changed by UPDATE 469
INCLUDE statement 421
REXX 644
SQLCA (SQL communications area)
Java 627
SQLCABC field of SQLCA 525
SQLCAID field of SQLCA 525

SQLCCSID field of SQLDA
in REXX 645
SQLCODE
description 265
field description 525
in REXX 644
SQLD field of SQLDA
field description 530
in REXX 645
information generated by DESCRIBE statement 392
SQLDA (SQL descriptor area)
C 593
COBOL 609
contents 529
DESCRIBE statement 391
description 529
FETCH statement 406
INCLUDE statement 421
Java 627
REXX 644
SQLDABC field of SQLDA 392, 530
SQLDAID field of SQLDA 391, 530
SQLDATA field of SQLDA
CCSID values 535
field description 532
in REXX 646
SQLDATALEN field of SQLDA 532
SQLERRD field of SQLCA 526, 644
SQLERRMC field of SQLCA 525, 644
SQLERRML field of SQLCA 525
SQLERROR clause of WHENEVER statement 475
SQLERRP field of SQLCA 525, 644
SQLIND field of SQLDA 532
field description 532
in REXX 646
SQLJ
basic concepts 627
comment 628
connecting to a data source 629
description 627
error handling 639
executable clause 627
format of SQL statement 627
importing Java packages 627
including code to access 627
loading JDBC driver 627
SQLJ iterator 632
valid SQL statements 627
SQLJ application
writing 627
SQLJ iterator
description 632
positioned iterator 635
retrieving rows in SQLJ 632, 635, 636
SQLLEN field of SQLDA 532
field description 532
in REXX 645
SQLLONGLEN field of SQLDA 532
SQLN field of SQLDA 391, 530
SQLNAME field of SQLDA 532
CCSID values 535
field description 532
in REXX 645
SQLPRECISION field of SQLDA 645
SQLSCALE field of SQLDA 645
SQLSTATE
description 265
field description 527
in REXX 644
values 539
SQLTYPE field of SQLDA 532
field description 532
in REXX 645
SQLVAR field of SQLDA 392, 532
SQLWARN field of SQLCA 526, 644
SQLWARNING clause of WHENEVER statement 475
SQRT function 211
STAFF sample table 679
statement-name
description 35
in DECLARE CURSOR statement 381, 382
in DESCRIBE
in C 593
in COBOL 610
in DESCRIBE statement 391
in EXECUTE statement 401
in OPEN statement 429
in PREPARE
in C 594
in PREPARE statement 433
length 36
states
connection 21
STATIC DISPATCH 336
STATIC DISPATCH clause
CREATE FUNCTION statement 319
static select 264
static SQL
definition 1
use of SQL path 38
STDDEV function 139
storage
assignment 60
storage structures 28
string
assignment 60
binary 46
character 44
columns 44, 46
comparison 65
constant
character 73
graphic 74
conversion 23
graphic 45
limitations on use of 48
LOB 47
variable
fixed-length 44
varying-length 44
string limits 510, 511
subquery
description 83, 234
HAVING clause 245
WHERE clause 243
subquery in a basic predicate
equivalent term 690
subselect
description 83, 234
equivalent term 690
subselect *(continued)*
examples of 246
in basic predicate 113
in CREATE VIEW statement 234, 376
in EXISTS predicate 117
in GROUP BY clause 244
in HAVING clause 245
in IN predicate 118
in quantified predicate 114
in UPDATE statement 468
in WHERE clause 243
substitution character 24
SUBSTR function 212
SUM function 140
synonym
CREATE ALIAS statement 303
synonym for qualifying a column name 79
syntax diagrams x
system schema 3

T

table
alias 303
changing 267
column 3
creating 353
dependent 4
descendent 4
description 3
designator 82
dropping 398
parent 4
primary key 3
relational database 1
renaming
RENAME statement 442
result table 3
row 3
self-referencing 4
temporary 431
table check constraint 6
TABLE clause
COMMENT statement 292
DROP statement 398
table expression
equivalent term 689
table-name
description 35
in ALTER TABLE statement 267, 270
in COMMENT statement 289, 292
in CREATE INDEX statement 338
in CREATE TABLE statement 353, 356
in DELETE statement 386, 387
in DROP statement 395, 398
in GRANT (Table or View Privileges) statement 418, 419
in INSERT statement 423, 424
in LOCK TABLE statement 428
in REVOKE (Table and View Privileges) statement 452, 453
in SELECT clause 235
in UPDATE statement 466, 467
unqualified, length of 36
TAN function 214

target specification
equivalent term 689
temporary tables in OPEN 431
time
arithmetic operations 103
duration 101
strings 50
TIME
assignment 62
data type 49, 359
function 215
timestamp
arithmetic operations 104
duration 101
strings 51
TIMESTAMP
assignment 63
data type 49, 359
function 216
TO
clause of CONNECT (Type 2) statement 300
tokens
delimiter 30
ordinary 30
SQL 30
transaction
equivalent term 689
TRANSLATE function 218
trigger
creating 368
dropping 398
TRIGGER clause
COMMENT statement 292
DROP statement 398
trigger-name
description 35
in COMMENT statement 289
in CREATE TRIGGER statement 369
in DROP statement 398
TRUNCATE function 220
truncation of numbers 59
truth table 126
truth valued logic 126
type
dropping 396
TYPE clause
DROP statement 396

U

UCASE function 222
UDF (user-defined function) 91
external 91
sourced 91
SQL 91
unary
minus 97
plus 97
uncommitted read 15
unconnected state 22
undefined reference 82
Unicode data
description 45, 46
UNION
result data type 68
UNION ALL operator of fullselect 248
UNION operator
duplicate rows 248
fullselect 248
UNIQUE clause
ALTER TABLE statement 274
CREATE INDEX statement 338
CREATE TABLE statement 362
UNIQUE clause of CREATE TABLE statement 273, 361
unique index 3
unique key 3
unique-constraint clause of CREATE TABLE statement 361
unit of work
description 12
ending 294, 455
updatable
view 379
UPDATE
clause of GRANT (Table or View Privileges) statement 419
clause of REVOKE (Table and View Privileges) statement 453
clause of select-statement 254
rules 469
statement 466
use in update-clause 254
UPDATE clause
select-statement 254
UPPER function 223
UR (uncommitted read) isolation level 15
USA (IBM USA standard)
argument in CHAR function 150
USAGE
in GRANT (Distinct Type Privileges) statement 410
in REVOKE (Distinct Type Privileges) statement 444
USAGE privilege 410, 444
USER special register 78
user-defined function 91
CREATE FUNCTION (external scalar) statement 314
CREATE FUNCTION (sourced) statement 325
CREATE FUNCTION (SQL scalar) statement 332
CREATE FUNCTION statement 310
external 91
sourced 91
SQL 91
user-defined type
description 51
user-defined types (UDTs)
casting 54
USING clause
CALL statement 284
EXECUTE statement 401
FETCH statement 406
OPEN statement 429

V

valid SQL statements
SQLJ 627
value expression
equivalent term 689
VALUE function 224
value in SQL 42
VALUES
statement 472
VALUES clause
VALUES INTO statement 473
VALUES clause of INSERT statement 424
VALUES INTO
statement 473
VAR function 141
VARCHAR
function 225
VARCHAR data type in CREATE TABLE statement 357
VARGRAPHIC
data type 358
function 227
variable names used in SQL 32
variables, host
C 597
COBOL 612
REXX 648
VARIANCE function 141
varying-length string 44, 46
view
alias 303
creating 376
deletable 379
description 8
dropping 399
insertable 379
name 35
read-only 379
updatable 379
VIEW clause
CREATE VIEW statement 376
DROP statement 398
view-name
description 35
in COMMENT statement 289, 292
in CREATE VIEW statement 376
in DELETE statement 386, 387
in DROP statement 395, 398
in GRANT (Table or View Privileges) statement 418, 419
in INSERT statement 423, 424
in REVOKE (Table and View Privileges) statement 452, 453
in SELECT clause 235
in UPDATE statement 466, 467
unqualified, length of 36

W

warning return code 264, 539
WEEK function 230
WEEK_ISO function 231
WHENEVER statement 475
C 597
COBOL 612
REXX, substitute for 647
WHERE clause
DELETE statement 387
SELECT INTO statement 457

WHERE clause *(continued)*
subselect 243
UPDATE statement 469
WHERE CURRENT OF clause
DELETE statement 388
UPDATE statement 469
WHILE statement 506
WITH CHECK OPTION
See CHECK OPTION clause of CREATE VIEW statement
WITH CHECK OPTION clause of CREATE VIEW statement
INSERT rules 426
UPDATE rules 470
WITH clause
DELETE statement 388, 469
INSERT statement 425
WITH GRANT OPTION clause
GRANT (Distinct Type Privileges) statement 411
GRANT (Function or Procedure Privileges) statement 415
GRANT (package privileges) statement 417
GRANT (Table or View Privileges) statement 420
WITH HOLD clause of DECLARE CURSOR statement 381
with positioned iterators 635
WITH RETURN clause of DECLARE CURSOR statement 382
with SQLJ iterators 632
WORK keyword
COMMIT statement 294
ROLLBACK statement 455

Y

YEAR function 232
YEAR labeled duration 96, 100
YEARS labeled duration 96, 100

IBM®

Printed in U.S.A.