

DB2 10 for z/OS

*Application Programming and SQL
Guide*

IBM

DB2 10 for z/OS

*Application Programming and SQL
Guide*



Note

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

Eleventh edition (August 2014)

This edition applies to DB2 10 for z/OS (product number 5605-DB2), DB2 10 for z/OS Value Unit Edition (product number 5697-P31), and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

© Copyright IBM Corporation 1983, 2014.

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

Contents

About this information	xv
Who should read this information.	xv
DB2 Utilities Suite	xv
Terminology and citations	xv
Accessibility features for DB2 10 for z/OS	xvi
How to send your comments	xvii
How to read syntax diagrams.	xvii
Chapter 1. Planning for and designing DB2 applications	1
Application and SQL release incompatibilities for migration from Version 8	1
Changes in BIND PACKAGE and BIND PLAN defaults (change introduced in Version 9.1)	2
Plan for the XML data type (change introduced in Version 9.1)	2
Changes to XMLNAMESPACES (change introduced in Version 9.1)	2
Changes to serialization of empty elements (change introduced in Version 9.1)	2
Availability of LOB or XML values in JDBC or SQLJ applications with progressive streaming (change introduced in Version 9.1)	2
Adjust applications that depend on error information that is returned from DB2-supplied stored procedures (change introduced in Version 9.1)	3
Some materialized query tables should be dropped (change introduced in Version 9.1)	3
Fully define objects (change introduced in Version 9.1)	3
Changes to PL/I applications with no DECLARE VARIABLE statements (change introduced in Version 9.1)	4
Automatic rebind of plans and packages created before DB2 Version 6	4
IBMREQD is no longer reliable as a release dependency mark.	4
Changed behavior for ODBC data conversion for the SQL_BINARY type (change introduced in Version 9.1)	4
Changed behavior of the INSERT statement with the OVERRIDING USER VALUE clause (change introduced in Version 9.1).	4
DESCRIBE no longer returns LONG type values (change introduced in Version 9.1)	5
DB2 enforces the restrictions about where a host variable array can be specified (change introduced in Version 9.1)	5
DEBUGSESSION system privilege required for continued debugging of SQL procedures (change introduced in Version 9.1).	5
Changes to the result length of the DECRYPT function (change introduced in Version 9.1)	5
Changes to the result length and data type of some functions (change introduced in Version 9.1)	5
COLTYPE column in SYSIBM.SYSCOLUMNS and SYSIBM.SYSCOLUMNS_HIST for LONG column types (change introduced in Version 9.1)	6
CREATEDBY column in SYSIBM.SYSDATATYPES, SYSIBM.SYSROUTINES, SYSIBM.SYSSEQUENCES, and SYSIBM.SYSTRIGGERS (change introduced in Version 9.1)	6
DB2 returns all DSNWZP output in the same format as DB2 parameters (change introduced in Version 9.1)	6
DB2 enforces the restriction that row IDs are not compatible with character strings when they are used with a set operator (change introduced in Version 9.1)	6
Database privileges on the DSNDB04 database now give you those privileges on all implicitly created databases (change introduced in Version 9.1)	6
Implicitly created objects that are associated with LOB columns require additional privileges (change introduced in Version 9.1)	7
Adjust applications to use LRHCLR instead of LGDISCLR (change introduced in Version 9.1)	7
Changed behavior for the CREATE statement (change introduced in Version 9.1)	7
The DECLARE statement and the work file database (change introduced in Version 9.1)	7
Adjust monitor programs that access OP buffers (change introduced in Version 9.1)	7
Changed behavior for system-required objects (change introduced in Version 9.1)	8
Changes to INSERT, UPDATE, or DELETE statements on some indexes (change introduced in Version 9.1).	9
LOBs with a maximum length greater than 1 GB can now be logged (change introduced in Version 9.1)	9
DB2 returns an error when a LOB value is specified for an argument to a stored procedure and the argument value is longer than the target parameter and the excess is not trailing blanks (change introduced in Version 9.1)	9
Changes to string formatting of decimal data	9
Change to maximum result length of VARCHAR function.	11

Leading or trailing blanks from the VARCHAR_FORMAT function format string are no longer removed (change introduced in Version 9.1)	11
Changes to VARCHAR_FORMAT function length attribute	11
New format element for VARCHAR_FORMAT function	11
Change to result of VARCHAR_FORMAT function with “HH12” format element	11
Change to result of VARCHAR_FORMAT function with “J” format element	12
New supported data types for VARCHAR_FORMAT function	12
Change to maximum result length of REPEAT function.	12
Change to maximum result length of XMLTABLE function	12
DB2 drops certain indexes when a unique constraint is dropped (change introduced in Version 9.1).	12
DB2 enforces restriction on specifying a CAST FROM clause for some forms of CREATE FUNCTION statements (change introduced in Version 9.1)	12
DB2 enforces restrictions on specifying the AS LOCATOR clause and TABLE LIKE clause (change introduced in Version 9.1)	13
DB2 enforces restriction on the CCSID parameter for the DECRYPT_BIT and DECRYPT_BINARY functions (change introduced in Version 9.1).	13
Changed behavior of CREATE PROCEDURE for an SQL procedure (change introduced in Version 9.1).	13
Explicitly qualify names of variables, parameters, and columns in SQL procedures (change introduced in Version 9.1)	13
Make any necessary program changes for possibly different values for RETURNED_SQLSTATE and DB2_RETURNED_SQLCODE (change introduced in Version 9.1)	14
SQLSTATE and SQLCODE SQL variables after a GET DIAGNOSTICS statement (change introduced in Version 9.1)	14
Coding multiple SQL statements in a handler body (change introduced in Version 9.1)	14
Unhandled warnings (change introduced in Version 9.1)	14
Change your programs to handle any changed messages from SQL procedures (change introduced in Version 9.1)	14
Enhanced data type checking for zero-length characters (change introduced in Version 9.1).	15
Adding a column generates a new table space version (change introduced in Version 9.1)	15
You cannot add a column and issue SELECT, INSERT, UPDATE, or DELETE statements in the same commit scope	15
CAST FROM clause of CREATE FUNCTION statement for SQL functions is no longer supported (change introduced in Version 9.1)	16
Specifying ALTER DATABASE STOGROUP for work file databases (change introduced in Version 9.1).	16
DB2 enforces restrictions about where an INTO clause can be specified (change introduced in Version 9.1)	16
Change to how a positive, signed integer in an ORDER BY clause is treated.	16
Binding DBRMs directly into plans is no longer supported	17
Some BIND PLAN and REBIND PLAN command options are no longer supported	17
Plans and packages should be converted to DRDA protocol	17
Change to GRANT statement	17
Change to IMMEDIATE option of BIND PACKAGE command	17
Changes to conversion of special characters in collection IDs and package names	17
Changes to the RELEASE bind option	18
Database metadata stored procedures are converted to Unicode	18
AUTHID is the default owner of packages that are bound by DSNTRIN	18
New default DEFINE attribute for dependent objects	19
Change for creating partitioned table spaces	19
Change to default for CREATE TABLESPACE statements	19
Upgrade to supported COBOL and PL/I compilers	20
GRAPHIC and NOGRAPHIC SQL processing options are removed.	20
SELECT FROM data change statements in BEFORE triggers no longer supported	20
RETURN statement in scalar functions must follow <i>option-list</i>	20
Changes to ROUND_TIMESTAMP and TRUNC_TIMESTAMP functions	21
Changes to result of NEXT_DAY function	21
Changes to MONTHS_BETWEEN function	21
Changes to TIMESTAMPDIFF function	21
Static SQL applications that use parallelism.	21
Enforced SELECT authorization checking for UPDATE and DELETE statements	22
Increased limit for work file record length	22
New restrictions for EXPLAIN tables	22
MEMBER CLUSTER table spaces indicated by MEMBER_CLUSTER column	22

Changed values for the modification level in the product signature	23
Changed behavior for the CREATE FUNCTION statement	23
Different SQLSTATE returned for some DELETE or UPDATE statements	23
Changed default behavior of multiple-row inserts for ODBC z/OS applications	23
Changes to ALTER TABLESPACE statement error codes	23
Change to CREATE and ALTER statements	24
Change to DESCRIBE statement	24
New restrictions on using DSNTIAUL	24
Changes to SYSROUTINES	24
Catalog restructured	24
Changed data type for an untyped parameter marker	24
Changes to handling of special values Infinity, sNaN, and NaN	24
Changes for INSTEAD OF triggers	25
Change to positioned update or delete statements	25
Change to stored procedure parameter values returned to non-Java clients	25
Change to results of JDBC method PreparedStatement.setTimestamp	26
Change in how DB2 returns stored procedure output parameter data to remote clients	26
Changes to datetime built-in functions	29
SQLCODE change for subsequent CAF CONNECT attempts	29
Change to serialization of an empty XML element	30
Data types of output arguments from a stored procedure call in a Java application	30
Change to IBM Data Server Driver for JDBC and SQLJ handling of TIMESTAMP WITH TIME ZONE data type	31
Delimiters used for accessing tables on DB2 for Linux, UNIX, and Windows	31
Qualify user-defined function names	32
SQLCODE changes	32
SQL reserved words	32
Application and SQL release incompatibilities for migration from Version 9.1	32
Automatic rebind of plans and packages created before DB2 Version 6	32
IBMREQD is no longer reliable as a release dependency mark	33
Changes to string formatting of decimal data	33
Change to maximum result length of VARCHAR function	34
Changes to VARCHAR_FORMAT function length attribute	34
New format element for VARCHAR_FORMAT function	35
Change to result of VARCHAR_FORMAT function with "HH12" format element	35
Change to result of VARCHAR_FORMAT function with "J" format element	35
New supported data types for VARCHAR_FORMAT function	35
Change to maximum result length of REPEAT function	35
Change to maximum result length of XMLTABLE function	35
Change to how a positive, signed integer in an ORDER BY clause is treated	36
Binding DBRMs directly into plans is no longer supported	36
Some BIND PLAN and REBIND PLAN command options are no longer supported	36
Plans and packages should be converted to DRDA protocol	36
Change to GRANT statement	37
Change to IMMEDIATE option of BIND PACKAGE command	37
Changes to conversion of special characters in collection IDs and package names	37
Changes to the RELEASE bind option	37
Database metadata stored procedures are converted to Unicode	37
AUTHID is the default owner of packages that are bound by DSNTRIN	38
New default DEFINE attribute for dependent objects	38
Change for creating partitioned table spaces	39
Change to default for CREATE TABLESPACE statements	39
Change to default SEGSIZE value for universal table spaces	39
Upgrade to supported COBOL and PL/I compilers	39
GRAPHIC and NOGRAPHIC SQL processing options are removed	40
SELECT FROM data change statements in BEFORE triggers no longer supported	40
RETURN statement in scalar functions must follow <i>option-list</i>	40
Changed behavior of LOCATE_IN_STRING function	40
Changes to ROUND_TIMESTAMP and TRUNC_TIMESTAMP functions	40
Changes to result of NEXT_DAY function	41
Changes to MONTHS_BETWEEN function	41
Changes to TIMESTAMPDIFF function	41

	Static SQL applications that use parallelism.	41
	Enforced SELECT authorization checking for UPDATE and DELETE statements	41
	Increased limit for work file record length	42
	New restrictions for EXPLAIN tables.	42
	MEMBER CLUSTER table spaces indicated by MEMBER_CLUSTER column	42
	Changed values for the modification level in the product signature.	42
	Changed behavior for the CREATE FUNCTION statement	43
	Different SQLSTATE returned for some DELETE or UPDATE statements	43
	Changed default behavior of multiple-row inserts for ODBC z/OS applications	43
	Changes to ALTER TABLESPACE statement error codes	43
	Change to CREATE and ALTER statements.	44
	Change to ALTER PROCEDURE statement.	44
	Change to DESCRIBE statement	44
	New restrictions on using DSNTIAUL	44
	Changes to SYSROUTINES	44
	Catalog restructured	44
	Changed data type for an untyped parameter marker	44
	Changes to handling of special values Infinity, sNaN, and NaN	44
	Changes for INSTEAD OF triggers	44
	Change to positioned update or delete statements	45
	Change to stored procedure parameter values returned to non-Java clients	45
	Change to results of JDBC method PreparedStatement.setTimestamp	46
	Change to behavior of comma operator in XQuery path expression.	46
	Change in how DB2 returns stored procedure output parameter data to remote clients	46
	Change to IBM Data Server Driver for JDBC and SQLJ handling of TIMESTAMP WITH TIME ZONE data type	49
	Changes to datetime built-in functions	49
	SQLCODE change for subsequent CAF CONNECT attempts	50
	Delimiters used for accessing tables on DB2 for Linux, UNIX, and Windows	50
	Qualify user-defined function names	51
	SQLCODE changes	51
	SQL reserved words	51
	Determining the value of any SQL processing options that affect the design of your program	51
	Changes that invalidate packages	52
	Determining the value of any bind options that affect the design of your program	53
	Programming applications for performance.	54
	Designing your application for recovery.	55
	Unit of work in TSO	56
	Unit of work in CICS	57
	Planning for program recovery in IMS programs	58
	Undoing selected changes within a unit of work by using savepoints	65
	Planning for recovery of table spaces that are not logged	66
	Designing your application to access distributed data	67
	Remote servers and distributed data	68
	Preparing for coordinated updates to two or more data sources	69
	Forcing restricted system rules in your program	69
	Creating a feed in IBM Mashup Center with data from a DB2 for z/OS server	70
 Chapter 2. Connecting to DB2 from your application program		73
	Invoking the call attachment facility	74
	Call attachment facility	77
	Making the CAF language interface (DSNALI) available	80
	Requirements for programs that use CAF	81
	How CAF modifies the content of registers.	82
	Implicit connections to CAF	82
	CALL DSNALI statement parameter list.	83
	Summary of CAF behavior	85
	CAF connection functions	86
	Turning on a CAF trace	98
	CAF return codes and reason codes	98
	Sample CAF scenarios	99
	Examples of invoking CAF	100

Invoking the Resource Recovery Services attachment facility	106
Resource Recovery Services attachment facility	108
Making the RRSAF language interface (DSNRLI) available	112
Requirements for programs that use RRSAF	114
How RRSAF modifies the content of registers	114
Implicit connections to RRSAF.	114
CALL DSNRLI statement parameter list	115
Summary of RRSAF behavior	116
RRSAF connection functions	118
RRSAF return codes and reason codes	148
Sample RRSAF scenarios	149
Program examples for RRSAF.	151
Universal language interface	153
Link-editing an application with DSNULI	155
Controlling the CICS attachment facility from an application	155
Detecting whether the CICS attachment facility is operational	156
Improving thread reuse in CICS applications.	157
Chapter 3. Coding SQL statements in application programs: General information	159
Declaring table and view definitions.	160
DCLGEN (declarations generator)	161
Generating table and view declarations by using DCLGEN	161
Including declarations from DCLGEN in your program	169
Example: Adding DCLGEN declarations to a library	169
Defining the items that your program can use to check whether an SQL statement executed successfully.	173
Defining SQL descriptor areas.	173
Declaring host variables and indicator variables.	174
Host variables	174
Host variable arrays	175
Host structures.	175
Indicator variables, arrays, and structures	176
Setting the CCSID for host variables.	178
Determining what caused an error when retrieving data into a host variable	179
Accessing an application defaults module	180
Compatibility of SQL and language data types	180
Embedding SQL statements in your application	183
Delimiting an SQL statement	183
Rules for host variables in an SQL statement	183
Retrieving a single row of data into host variables	184
Determining whether a retrieved value in a host variable is null or truncated	186
Determining whether a column value is null	188
Updating data by using host variables	189
Inserting a single row by using a host variable	189
Inserting null values into columns by using indicator variables or arrays	190
Host variable arrays in an SQL statement	191
Retrieving multiple rows of data into host variable arrays	192
Inserting multiple rows of data from host variable arrays.	192
Retrieving a single row of data into a host structure	193
Including dynamic SQL in your program	193
Checking the execution of SQL statements.	227
Checking the execution of SQL statements by using the SQLCA	228
Checking the execution of SQL statements by using SQLCODE and SQLSTATE	232
Checking the execution of SQL statements by using the WHENEVER statement	233
Checking the execution of SQL statements by using the GET DIAGNOSTICS statement	234
Handling SQL error codes	239
Arithmetic and conversion errors.	240
Writing applications that enable users to create and modify tables.	240
Saving SQL statements that are translated from user requests	240
XML data in embedded SQL applications	241
Host variable data types for XML data in embedded SQL applications	241
XML column updates in embedded SQL applications	246

XML data retrieval in embedded SQL applications	248
Programming examples	251
Examples of programs that call stored procedures	252

Chapter 4. Coding SQL statements in assembler application programs 253

Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler	253
Defining SQL descriptor areas in assembler	254
Declaring host variables and indicator variables in assembler	254
Host variables in assembler	255
Indicator variables in assembler	260
Equivalent SQL and assembler data types	261
SQL statements in assembler programs	266
Delimiters in SQL statements in assembler programs	271
Macros for assembler applications	271
Programming examples in assembler	271

Chapter 5. Coding SQL statements in C application programs 273

Defining the SQL communications area, SQLSTATE, and SQLCODE in C	273
Defining SQL descriptor areas in C	274
Declaring host variables and indicator variables in C	275
Host variables in C	275
Host variable arrays in C	287
Host structures in C	295
Indicator variables, indicator arrays, and host structure indicator arrays in C	297
Referencing pointer host variables in C programs	298
Declaring pointer host variables in C programs	300
Equivalent SQL and C data types.	301
SQL statements in C programs	307
Delimiters in SQL statements in C programs	311
Programming examples in C	311

Chapter 6. Coding SQL statements in COBOL application programs. 323

Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL	323
Defining SQL descriptor areas in COBOL	324
Declaring host variables and indicator variables in COBOL	325
Host variables in COBOL	326
Host variable arrays in COBOL	335
Host structures in COBOL	344
Indicator variables, indicator arrays, and host structure indicator arrays in COBOL	349
Controlling the CCSID for COBOL host variables	351
Equivalent SQL and COBOL data types	352
SQL statements in COBOL programs	357
Delimiters in SQL statements in COBOL programs	363
Object-oriented extensions in COBOL	363
Programming examples in COBOL	363

Chapter 7. Coding SQL statements in Fortran application programs. 395

Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran	395
Defining SQL descriptor areas in Fortran	396
Declaring host variables and indicator variables in Fortran	397
Host variables in Fortran	397
Indicator variables in Fortran	400
Equivalent SQL and Fortran data types.	401
SQL statements in Fortran programs.	403
Delimiters in SQL statements in Fortran programs	406

Chapter 8. Coding SQL statements in PL/I application programs 407

Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I	407
Defining SQL descriptor areas in PL/I	408
Declaring host variables and indicator variables in PL/I	408

Host variables in PL/I	409
Host variable arrays in PL/I	415
Host structures in PL/I	420
Indicator variables in PL/I	422
Equivalent SQL and PL/I data types	423
SQL statements in PL/I programs	427
Delimiters in SQL statements in PL/I programs	432
Programming examples in PL/I	432
Chapter 9. Coding SQL statements in REXX application programs	437
Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX	437
Defining SQL descriptor areas in REXX.	437
Equivalent SQL and REXX data types	438
SQL statements in REXX programs	439
Delimiters in SQL statements in REXX programs	442
Accessing the DB2 REXX language support application programming interfaces	442
Ensuring that DB2 correctly interprets character input data in REXX programs	444
Passing the data type of an input data type to DB2 for REXX programs	445
Setting the isolation level of SQL statements in a REXX program	445
Retrieving data from DB2 tables in REXX programs	446
Cursors and statement names in REXX.	447
Programming examples in REXX	448
Chapter 10. Creating and modifying DB2 objects	459
Creating tables	459
Data types	460
Storing LOB data in a table.	463
Identity columns	466
Creating tables for data integrity	468
Creating work tables for the EMP and DEPT sample tables	478
Creating created temporary tables	479
Creating declared temporary tables	481
Providing a unique key for a table	483
Fixing tables with incomplete definitions	484
Dropping tables	484
Defining a view	485
Views	486
Dropping a view	487
Creating a common table expression.	487
Common table expressions	488
Examples of recursive common table expressions	489
Creating triggers	493
Invoking a stored procedure or user-defined function from a trigger	501
Inserting, updating, and deleting data in views by using INSTEAD OF triggers	503
Trigger packages	504
Trigger cascading	505
Order of multiple triggers	505
Interactions between triggers and referential constraints	506
Interactions between triggers and tables that have multilevel security with row-level granularity	508
Triggers that return inconsistent results.	508
Sequence objects	511
DB2 object relational extensions	512
Creating a distinct type	513
Distinct types	513
Example of distinct types, user-defined functions, and LOBs	514
Defining a user-defined function	517
User-defined functions	520
Components of a user-defined function definition	523
Writing an external user-defined function	525
Making a user-defined function reentrant	547

Special registers in a user-defined function or a stored procedure	548
Accessing transition tables in a user-defined function or stored procedure	551
Preparing an external user-defined function for execution	554
Abnormal termination of an external user-defined function	554
Saving information between invocations of a user-defined function by using a scratchpad	555
Example of creating and using a user-defined scalar function	556
User-defined function samples that ship with DB2	557
Creating a stored procedure	558
Stored procedures	559
Creating a native SQL procedure	572
Migrating an external SQL procedure to a native SQL procedure	596
Changing an existing version of a native SQL procedure	598
Regenerating an existing version of a native SQL procedure	599
Removing an existing version of a native SQL procedure	599
Creating an external SQL procedure	600
Creating an external stored procedure	615
Creating multiple versions of external procedures and external SQL procedures	654

Chapter 11. Adding and modifying data 655

Inserting data into tables	655
Inserting rows by using the INSERT statement	655
Inserting data and updating data in a single operation	661
Selecting values while inserting data	663
Adding data to the end of a table	670
Storing data that does not have a tabular format	670
Updating table data	670
Selecting values while updating data	672
Updating thousands of rows	672
Deleting data from tables	673
Selecting values while deleting data	674

Chapter 12. Accessing data 677

Determining which tables you have access to	677
Displaying information about the columns for a given table	677
Retrieving data by using the SELECT statement	678
Selecting derived columns	681
Selecting XML data	681
Formatting the result table	682
Combining result tables from multiple SELECT statements	688
Summarizing group values	692
Finding rows that were changed within a specified period of time	694
Joining data from more than one table	695
Optimizing retrieval for a small set of rows	706
Creating recursive SQL by using common table expressions	707
Updating data as it is retrieved from the database	708
Avoiding decimal arithmetic errors	708
Implications of using SELECT *	709
Subqueries	710
Restrictions when using distinct types with UNION, EXCEPT, and INTERSECT	719
Comparison of distinct types	719
Nested SQL statements	720
Retrieving a set of rows by using a cursor	722
Cursors	722
Accessing data by using a row-positioned cursor	726
Accessing data by using a rowset-positioned cursor	731
Retrieving rows by using a scrollable cursor	737
Accessing XML or LOB data quickly by using FETCH WITH CONTINUE	742
Determining the attributes of a cursor by using the SQLCA	745
Determining the attributes of a cursor by using the GET DIAGNOSTICS statement	746
Scrolling through previously retrieved data	746

Updating previously retrieved data	748
FETCH statement interaction between row and rowset positioning	748
Examples of fetching rows by using cursors	749
Specifying direct row access by using row IDs	754
ROWID columns	756
Ways to manipulate LOB data	756
LOB host variable, LOB locator, and LOB file reference variable declarations	757
LOB and XML materialization	762
Saving storage when manipulating LOBs by using LOB locators	763
Deferring evaluation of a LOB expression to improve performance	765
LOB file reference variables	767
Referencing a sequence object	769
Retrieving thousands of rows	770
Determining when a row was changed	770
Checking whether an XML column contains a certain value	771
Accessing DB2 data that is not in a table	771
Ensuring that queries perform sufficiently	772
Items to include in a batch DL/I program	772
Chapter 13. Invoking a user-defined function	777
Determining the authorization ID for invoking user-defined functions	779
Ensuring that DB2 executes the intended user-defined function	779
How DB2 resolves functions	780
Checking how DB2 resolves functions by using DSN_FUNCTION_TABLE	783
Restrictions when passing arguments with distinct types to functions	787
Cases when DB2 casts arguments for a user-defined function	788
Chapter 14. Calling a stored procedure from your application	791
Passing large output parameters to stored procedures by using indicator variables	796
Data types for calling stored procedures	797
Calling a stored procedure from a REXX procedure	797
Preparing a client program that calls a remote stored procedure	800
How DB2 determines which stored procedure to run	801
Calling different versions of a stored procedure from a single application	802
Invoking multiple instances of a stored procedure	803
Designating the active version of a native SQL procedure	804
Temporarily overriding the active version of a native SQL procedure	804
Specifying the number of stored procedures that can run concurrently	805
Retrieving the procedure status	806
Writing a program to receive the result sets from a stored procedure	807
DB2-supplied stored procedures	811
WLM_REFRESH stored procedure	816
WLM_SET_CLIENT_INFO stored procedure	820
DSN_WLM_APPLENV stored procedure	822
DSNACICS stored procedure	825
DSNAIMS stored procedure	833
DSNAIMS2 stored procedure	837
DSNACCOR stored procedure (deprecated)	842
XSR_REGISTER stored procedure	863
XSR_ADDSCHEMADOC stored procedure	865
XSR_COMPLETE stored procedure	866
XSR_REMOVE stored procedure	868
Chapter 15. Coding methods for distributed data	871
Accessing distributed data by using three-part table names	871
Accessing remote declared temporary tables by using three-part table names	873
Accessing distributed data by using explicit CONNECT statements	874
Specifying a location alias name for multiple sites	875
Releasing connections	875
Transmitting mixed data	876

Identifying the server at run time	876
SQL limitations at dissimilar servers.	876
Support for executing long SQL statements in a distributed environment	877
Distributed queries against ASCII or Unicode tables	877
Restrictions when using scrollable cursors to access distributed data	878
Restrictions when using rowset-positioned cursors to access distributed data	878
WebSphere MQ with DB2	879
WebSphere MQ messages	879
DB2 MQ functions and DB2 MQ XML stored procedures.	881
Generating XML documents from existing tables and sending them to an MQ message queue	883
Shredding XML documents from an MQ message queue	884
DB2 MQ tables	884
Basic messaging with WebSphere MQ	892
Sending messages with WebSphere MQ	893
Retrieving messages with WebSphere MQ	894
Application to application connectivity with WebSphere MQ	895
Asynchronous messaging in DB2 for z/OS	898
Chapter 16. DB2 as a web services consumer and provider	909
Deprecated: The SOAPHTTPV and SOAPHTTPC user-defined functions	909
The SOAPHTTPNV and SOAPHTTPNC user-defined functions	911
SQLSTATEs for DB2 as a web services consumer	912
Chapter 17. Preparing an application to run on DB2 for z/OS	915
Setting the DB2I defaults	917
Processing SQL statements	918
Processing SQL statements by using the DB2 precompiler	920
Processing SQL statements by using the DB2 coprocessor.	926
Translating command-level statements in a CICS program	929
Differences between the DB2 precompiler and the DB2 coprocessor	930
Options for SQL statement processing	932
Compiling and link-editing an application.	942
Binding an application	943
Binding a DBRM to a package.	944
Binding an application plan	949
Bind process for remote access	953
Binding a batch program	956
Conversion of DBRMs that are bound to a plan to DBRMs that are bound to a package	957
Converting an existing plan into packages to run remotely	958
Setting the program level	959
DYNAMICRULES bind option	959
Dynamic plan selection	961
Rebinding an application	962
Rebinding a package	963
Rebinding a plan	964
Rebinding lists of plans and packages	965
Generating lists of REBIND commands.	965
Automatic rebinding	970
Specifying the rules that apply to SQL behavior at run time.	972
DB2 program preparation overview	973
Input and output data sets for DL/I batch jobs	975
DB2-supplied JCL procedures for preparing an application	978
JCL to include the appropriate interface code when using the DB2-supplied JCL procedures	978
Tailoring DB2-supplied JCL procedures for preparing CICS programs	979
DB2I primary option menu.	981
DB2I panels that are used for program preparation.	982
DB2 Program Preparation panel	983
DB2I Defaults Panel 1	987
DB2I Defaults Panel 2	989
Precompile panel	990

Bind Package panel	993
Bind Plan panel	996
Defaults for Bind Package and Defaults for Rebind Package panels	998
Defaults for Bind Plan and Defaults for Rebind Plan panels	1001
System Connection Types panel	1003
Panels for entering lists of values	1004
Program Preparation: Compile, Link, and Run panel	1005
DB2I panels that are used to rebind and free plans and packages.	1007
Bind/Rebind/Free Selection panel	1008
Rebind Package panel	1009
Rebind Trigger Package panel	1011
Rebind Plan panel	1013
Free Package panel	1014
Free Plan panel	1015
Chapter 18. Running an application on DB2 for z/OS	1017
DSN command processor	1017
DB2I Run panel	1018
Running a program in TSO foreground	1019
Running a DB2 REXX application	1020
Invoking programs through the Interactive System Productivity Facility	1020
ISPF	1021
Invoking a single SQL program through ISPF and DSN	1022
Invoking multiple SQL programs through ISPF and DSN	1023
Loading and running a batch program	1024
Authorization for running a batch DL/I program	1025
Restarting a batch program	1025
Running stored procedures from the command line processor.	1028
Command line processor CALL statement	1028
Example of running a batch DB2 application in TSO	1029
Example of calling applications in a command procedure	1030
Chapter 19. Testing and debugging an application program on DB2 for z/OS	1031
Designing a test data structure	1031
Analyzing application data needs	1031
Authorization for test tables and applications	1033
Example SQL statements to create a comprehensive test structure	1033
Populating the test tables with data	1034
Methods for testing SQL statements	1034
Executing SQL by using SPUFI	1035
Content of a SPUFI input data set	1039
The SPUFI panel	1039
Changing SPUFI defaults	1041
Setting the SQL terminator character in a SPUFI input data set	1047
Controlling toleration of warnings in SPUFI.	1048
Output from SPUFI	1048
Testing an external user-defined function.	1050
Testing a user-defined function by using the Debug Tool for z/OS	1050
Testing a user-defined function by routing the debugging messages to SYSPRINT	1052
Testing a user-defined function by using driver applications	1052
Testing a user-defined function by using SQL INSERT statements	1052
Debugging stored procedures	1052
Debugging stored procedures with the Debug Tool and IBM VisualAge COBOL	1054
Debugging a C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS	1055
Debugging stored procedures by using the Unified Debugger	1055
Debugging stored procedures with the Debug Tool for z/OS	1056
Recording stored procedure debugging messages in a file	1058
Driver applications for debugging procedures	1059
DB2 tables that contain debugging information.	1059
Debugging an application program.	1059

Locating the problem in an application	1059
Techniques for debugging programs in TSO	1064
Techniques for debugging programs in IMS	1065
Techniques for debugging programs in CICS	1066
Finding a violated referential or check constraint	1070
Chapter 20. DB2 sample applications and data	1071
DB2 sample tables	1071
Activity table (DSN8A10.ACT)	1071
Department table (DSN8A10.DEPT)	1072
Employee table (DSN8A10.EMP)	1074
Employee photo and resume table (DSN8A10.EMP_PHOTO_RESUME)	1077
Project table (DSN8A10.PROJ)	1079
Project activity table (DSN8A10.PROJACT)	1080
Employee-to-project activity table (DSN8A10.EMPPROJACT)	1081
Unicode sample table (DSN8A10.DEMO_UNICODE)	1082
Relationships among the sample tables	1083
Views on the sample tables	1084
Storage of sample application tables	1088
DB2 sample applications	1092
Types of sample applications	1094
Application languages and environments for the sample applications	1096
Sample applications in TSO	1097
Sample applications in IMS	1100
Sample applications in CICS	1100
DSNTIAUL	1100
DSNTIAD	1106
DSNTEP2 and DSNTEP4	1108
Information resources for DB2 for z/OS and related products	1115
Notices	1117
Programming interface information.	1118
Trademarks.	1119
Privacy policy considerations.	1119
Glossary	1121
Index	1123

About this information

This information discusses how to design and write application programs that access DB2® for z/OS® (DB2), a highly flexible relational database management system (DBMS).

This information assumes that your DB2 subsystem is running in Version 10 new-function mode. Generally, new functions that are described, including changes to existing functions, statements, and limits, are available only in new-function mode, unless explicitly stated otherwise. Exceptions to this general statement include optimization and virtual storage enhancements, which are also available in conversion mode unless stated otherwise. In Versions 8 and 9, most utility functions were available in conversion mode. However, for Version 10, most utility functions work only in new-function mode.

Who should read this information

This information is for DB2 application developers who are familiar with Structured Query Language (SQL) and who know one or more programming languages that DB2 supports.

DB2 Utilities Suite

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

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

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

These informational APARs are periodically updated.

Related information

DB2 utilities packaging (Utility Guide)

Terminology and citations

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

The following terms are used as indicated:

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

Tivoli® OMEGAMON® XE

Refers to any of the following products:

- IBM® Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS

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

C, C++, and C language

Represent the C or C++ programming language.

CICS® Represents CICS Transaction Server for z/OS.

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

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

RACF®

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

Accessibility features for DB2 10 for z/OS

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

Accessibility features

The following list includes the major accessibility features in z/OS products, including DB2 10 for z/OS. These features support:

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

Tip: The Information Management Software for z/OS Solutions Information Center (which includes information for DB2 10 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

You can access DB2 10 for z/OS ISPF panel functions by using a keyboard or keyboard shortcut keys.

For information about navigating the DB2 10 for z/OS ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

Online documentation for DB2 10 for z/OS is available in the Information Management Software for z/OS Solutions Information Center, which is available at the following website: <http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/index.jsp>

IBM and accessibility

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

How to send your comments

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

- Send your comments by email to db2zinfo@us.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title or a help topic title).
- You can also send comments by using the **Feedback** link at the footer of each page in the Information Management Software for z/OS Solutions Information Center at <http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/index.jsp>.

How to read syntax diagrams

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

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

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

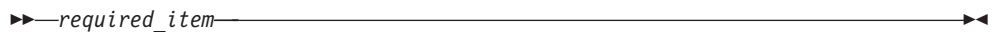
The \blacktriangleright — symbol indicates the beginning of a statement.

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

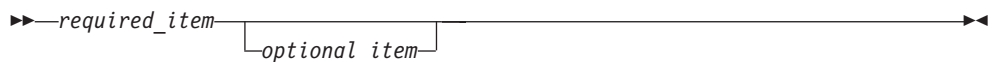
The \blacktriangleright — symbol indicates that a statement is continued from the previous line.

The — \blacktriangleleft symbol indicates the end of a statement.

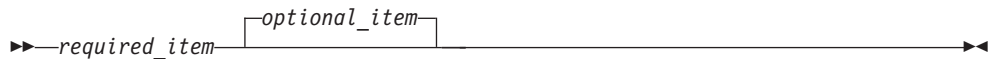
- Required items appear on the horizontal line (the main path).



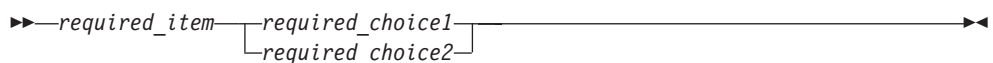
- Optional items appear below the main path.



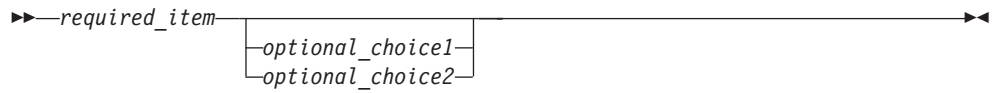
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.



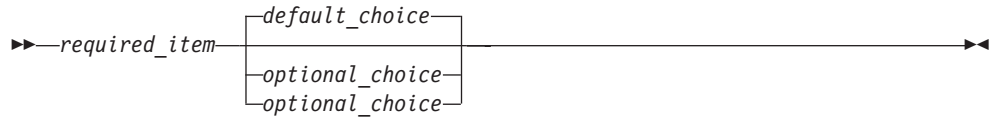
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



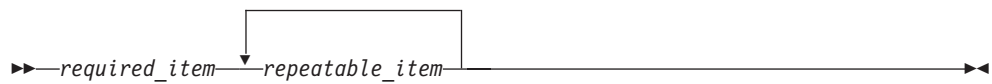
If choosing one of the items is optional, the entire stack appears below the main path.



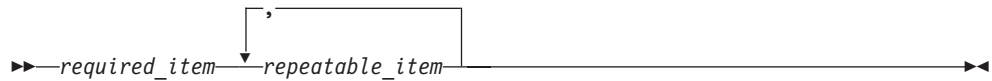
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



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



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



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

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



fragment-name:



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

Chapter 1. Planning for and designing DB2 applications

Before you write or run your program, you need to make some planning and design decisions. These decisions need to be made whether you are writing a new DB2 application or migrating an existing application from a previous release of DB2.





If you are migrating an existing application from a previous release of DB2, read the application and SQL release incompatibilities and make any necessary changes in the application.

If you are writing a new DB2 application, first determine the following items:


- the value of some of the SQL processing options
- the binding method
- the value of some of the bind options

Then make sure that your program implements the appropriate recommendations so that it promotes concurrency, can handle recovery and restart situations, and can efficiently access distributed data.

Related tasks:

-  Programming applications for performance (DB2 Performance)
-  Programming for concurrency (DB2 Performance)
-  Writing efficient SQL queries (DB2 Performance)
-  Improving performance for applications that access distributed data (DB2 Performance)

Related reference:

-  BIND and REBIND options for packages and plans (DB2 Commands)

Application and SQL release incompatibilities for migration from Version 8

When you migrate from DB2 Version 8 to Version 10, be aware of the application and SQL release incompatibilities.

GUPI

Plan for the following changes in Version 9.1 and Version 10 that might affect your migration.

Release incompatibilities that were changed or added since the first edition of this Version 10 publication are indicated by a vertical bar in the left margin. In other areas of this publication, a vertical bar in the margin indicates a change or addition that has occurred since the Version 9.1 release of this publication.

Changes in BIND PACKAGE and BIND PLAN defaults (change introduced in Version 9.1)

In DB2 Version 9, the default value for bind option CURRENTDATA is changed from YES to NO. This applies to the BIND PLAN and the BIND PACKAGE subcommands, the CREATE TRIGGER SQL statements for trigger packages, and the CREATE PROCEDURE and the ALTER PROCEDURE ADD VERSION SQL statements for SQL PL procedure packages. Specifying NO for CURRENTDATA is the best option for performance.

The default value for bind option ISOLATION is changed from RR to CS. This applies to the BIND PLAN and the remote BIND PACKAGE subcommands. For the BIND PACKAGE subcommand, the current default (plan value) stays. The default change does not apply to implicitly built CTs (for example, DISTSERV CTs).

If DBPROTOCOL(PRIVATE) is specified, the Version 10 BIND or REBIND command parser issues error message DSNT225I and fails the BIND or REBIND request to prevent any outbound private protocol communications as a requester. DSNT225I indicates that DBPROTOCOL(PRIVATE) is no longer a supported bind option.

All BIND statements for plans and packages that are bound during the installation or migration process specify the ISOLATION parameter explicitly, except for routines that do not fetch data. The current settings are maintained for compatibility.

Plan for the XML data type (change introduced in Version 9.1)

Drop any user-defined data types with the name "XML" to prevent problems with the new Version 9 built-in XML data type. You can re-create the existing user-defined data types with new names.

Changes to XMLNAMESPACES (change introduced in Version 9.1)

In DB2 Version 8, in the XMLNAMESPACES function, if the XML-namespace-uri argument has a value of <http://www.w3.org/XML/1998/namespace> or <http://www.w3.org/2000/xmlns/>, DB2 does not issue an error. Starting in Version 9 conversion mode, DB2 issues an error.

Changes to serialization of empty elements (change introduced in Version 9.1)

In Version 8, DB2 serializes empty XML elements in a different way than it serializes them in Version 10. In Version 8, empty element "a" is serialized as `<a>`. Starting in Version 9 conversion mode, empty element "a" is serialized as `<a/>`.

Availability of LOB or XML values in JDBC or SQLJ applications with progressive streaming (change introduced in Version 9.1)

In previous releases, if a JDBC or SQLJ application retrieves LOB data into an application variable, the contents of the application variable are still available after the cursor is moved or closed. Version 9 supports streaming. The IBM Data Server Driver for JDBC and SQLJ uses progressive streaming as the default for retrieval of

LOB or XML values. When progressive streaming is in effect, the contents of LOB or XML variables are no longer available after the cursor is moved or closed.

Adjust applications that depend on error information that is returned from DB2-supplied stored procedures (change introduced in Version 9.1)

Adjust any applications that call one of the following stored procedures and then check and process the specific SQLCODE or SQLSTATE that is returned by the CALL statement:

- SQLJ.INSTALL_JAR
- SQLJ.REMOVE_JAR
- SQLJ.REPLACE_JAR
- SQLJ.DB2_INSTALL_JAR
- SQLJ.DB2_REPLACE_JAR
- SQLJ.DB2_REMOVE_JAR
- SQLJ.DB2_UPDATEJARINFO

Starting in Version 9, these stored procedures return more meaningful SQLCODEs and SQLSTATEs than they return in previous releases of DB2. The other input and output parameters of these stored procedures have not changed.

For example, the following application needs to change because -20201 is no longer the SQLCODE that is returned. Successful execution (SQLCODE 0) is not affected.

```
CALL SQLJ.REMOVE_JAR(...)  
IF (SQLCODE = -20201) THEN  
DO;  
...  
END;
```

Some materialized query tables should be dropped (change introduced in Version 9.1)

Before migrating to conversion mode from Version 8, drop all materialized query tables that are based on the SYSIBM.SYSROUTINES catalog table. During migration to conversion mode from Version 8, if any materialized query tables are based on the SYSIBM.SYSROUTINES catalog table, SQLCODE -750 is issued.

Before migrating to enabling-new-function mode from Version 8, drop all materialized query tables that are based on the SYSIBM.SYSPACKSTMT catalog table. During migration to enabling-new-function mode from Version 8, if any materialized query tables are based on the SYSIBM.SYSPACKSTMT catalog table, SQLCODE -750 is issued.

Fully define objects (change introduced in Version 9.1)

Ensure that you do not have any incomplete object definitions in your DB2 Version 8 catalog. For example, if a table has a primary or unique key defined but the enforcing primary or unique key index does not exist, the table definition is considered incomplete. You need to complete or drop all such objects before you begin migration because their behavior will be different in Version 10. For example, if you attempt to create an enforcing primary key index to complete a table definition in Version 10 and the residing table space is implicitly created, the index is treated as a regular index instead of an enforcing index.

Changes to PL/I applications with no DECLARE VARIABLE statements (change introduced in Version 9.1)

For PL/I applications with no DECLARE VARIABLE statements, the rules for host variables and string constants in the FROM clause of a PREPARE or EXECUTE IMMEDIATE statement have changed. A host variable must be a varying-length string variable that is preceded by a colon. A PL/I string cannot be preceded by a colon.

Automatic rebind of plans and packages created before DB2 Version 6

If you have plans and packages that were bound before DB2 Version 6 and you specified YES or COEXIST in the AUTO BIND field of panel DSNTIPO, DB2 Version 10 autobinds these packages. Thus, you might experience an execution delay the first time that such a plan is loaded. Also, DB2 might change the access path due to the autobind, potentially resulting in a more efficient access path.

If you specify NO in the AUTO BIND field of panel DSNTIPO, DB2 Version 10 returns SQLCODE -908, SQLSTATE 23510 for each attempt to use such a package or plan until it is rebound.

IBMREQD is no longer reliable as a release dependency mark

The IBMREQD field in DB2 catalog tables is no longer a reliable indicator for determining release dependencies. Use the RELCREATED or RELBOUND fields instead.

Changed behavior for ODBC data conversion for the SQL_BINARY type (change introduced in Version 9.1)

In releases before Version 9.1, when ODBC applications used the SQL_BINARY type to bind parameter markers, ODBC mapped the SQL_BINARY type to CHAR FOR BIT DATA. In Version 10, when the DB2 server is in Version 10 new-function mode, ODBC maps SQL_BINARY to BINARY. Because CHAR FOR BIT DATA fields are padded with blanks, and BINARY fields are not padded, applications might experience differences in behavior.

For example, in releases before Version 9.1, if the target CHAR FOR BIT DATA column was shorter than the SQL_BINARY input host variable, and the truncated values were blanks, DB2 did not generate an error. In Version 10, if the target BINARY column is shorter than the SQL_BINARY input host variable, and the truncated values are hexadecimal zeroes, DB2 generates an error.

Changed behavior of the INSERT statement with the OVERRIDING USER VALUE clause (change introduced in Version 9.1)

When the INSERT statement is specified with the OVERRIDING USER VALUE clause, the value for the insert operation is ignored for columns that are defined with the GENERATED BY DEFAULT or GENERATED ALWAYS attribute.

DESCRIBE no longer returns LONG type values (change introduced in Version 9.1)

Because DB2 no longer stores LONG type values in the catalog, when you execute a DESCRIBE statement against a column with a LONG VARCHAR or LONG VARGRAPHIC data type, the DESCRIBE statement returns the values as VARCHAR or VARGRAPHIC data types.

DB2 enforces the restrictions about where a host variable array can be specified (change introduced in Version 9.1)

host-variable-array is the meta-variable for host variable arrays in syntax diagrams. *host-variable-array* is included only in the syntax for multi-row FETCH, multi-row INSERT, multi-row MERGE, and EXECUTE in support of a dynamic multi-row INSERT or MERGE statement. *host-variable-array* is not included in the syntax diagram for *expression*, so a host variable array cannot be used in other contexts. In previous releases, if you specified *host-variable-array* in an unsupported context, you received no errors. In Version 10, if a host variable array is referenced in an unsupported context, DB2 issues an error.

For more information about where you can specify the *host-variable-array* variable, see Host variable arrays in an SQL statement (DB2 Application programming and SQL).

DEBUGSESSION system privilege required for continued debugging of SQL procedures (change introduced in Version 9.1)

After you migrate to new-function mode, users that debug external SQL procedures need the DEBUGSESSION system privilege. (External SQL procedures were previously called SQL procedures in Version 8.) Only users of the new Unified Debugger enabled client platforms need this system privilege. Users of the Version 8 SQL Debugger-enabled client platforms do not need this system privilege.

Changes to the result length of the DECRYPT function (change introduced in Version 9.1)

The result length of the DECRYPT function is shortened to 8 bytes less than the length of the input value. If the result expands because of a difference between input and result CCSIDs, you must cast the encrypted data to a larger VARCHAR value before the DECRYPT function is run.

Changes to the result length and data type of some functions (change introduced in Version 9.1)

For the following built-in functions, if all parameters have data type DECFLOAT(*n*), the result has data type DECFLOAT(34):

- AVG
- STDDEV
- STDDEV_SAMP
- SUM
- VARIANCE
- VARIANCE_SAMP

For the following built-in functions, if the input is all integer or decimal values, or is a mixture of float or double values with integer or decimal values, the result data type is DOUBLE:

- CORRELATION
- COVARIANCE
- COVARIANCE_SAMP

COLTYPE column in SYSIBM.SYSCOLUMNS and SYSIBM.SYSCOLUMNS_HIST for LONG column types (change introduced in Version 9.1)

When new tables are created with LONG VARCHAR or LONG VARGRAPHIC columns, the COLTYPE values in SYSIBM.SYSCOLUMNS and SYSIBM.SYSCOLUMNS_HIST contain VARCHAR or VARG.

CREATEDBY column in SYSIBM.SYSDATATYPES, SYSIBM.SYSROUTINES, SYSIBM.SYSSEQUENCES, and SYSIBM.SYSTRIGGERS (change introduced in Version 9.1)

The CREATEDBY column might contain a different value than in previous releases of DB2. The column might contain a different value in static CREATE statements for distinct types, functions, and procedures or when a dynamic SQL statement sets the CURRENT SQLID value to a value other than USER.

DB2 returns all DSNWZP output in the same format as DB2 parameters (change introduced in Version 9.1)

In previous releases, DSNWZP returned the current setting of several system parameters in a format other than the one used by the system parameter macros. For example, DSN6SPRM expected the setting for EDMPOOL in kilobytes, and DSNWZP returned it in bytes. In Version 10, DB2 returns all DSNWZP output in the same format as DB2 parameters. Modify programs that call DSNWZP if they compensate for the format differences.

DB2 enforces the restriction that row IDs are not compatible with character strings when they are used with a set operator (change introduced in Version 9.1)

In previous releases, DB2 did not always enforce the restriction that row IDs are not compatible with character strings. In Version 10, DB2 enforces the restriction that row IDs are not compatible with string types when they are used with a set operator (UNION, INTERSECT, or EXCEPT).

Database privileges on the DSNDB04 database now give you those privileges on all implicitly created databases (change introduced in Version 9.1)

Because database privileges on the DSNDB04 database now give you those privileges on all implicitly created databases, careful consideration is needed before you grant database privileges on DSNDB04. For example, in Version 10, if you have the STOPDB privilege on DSNDB04, you also have the STOPDB privilege on all implicitly created databases.

Implicitly created objects that are associated with LOB columns require additional privileges (change introduced in Version 9.1)

In releases before Version 9.1, implicitly created objects that are associated with LOB columns do not require CREATETAB and CREATETS privileges on the database of the base table. Those implicitly created objects also do not require the USE privilege on the buffer pool and storage group that is used by the LOB objects. In Version 10, these privileges are required.

Adjust applications to use LRHCLR instead of LGDISCLR (change introduced in Version 9.1)

The LGDISCLR field in the DSNDQJ00 macro has been removed. Update applications that use the LGDISCLR value in the DSNDQJ00 mapping macro to determine whether a log record is a compensation log record to use the LRHCLR value instead.

Changed behavior for the CREATE statement (change introduced in Version 9.1)

You can no longer create databases with the AS TEMP clause or table spaces that specify TEMP as the target database. The TEMP database is no longer used by DB2. The WORKFILE database is the only temporary database.

The DECLARE statement and the work file database (change introduced in Version 9.1)

If you have applications in Version 8 that issue DECLARE SENSITIVE STATIC SCROLL CURSOR or DECLARE GLOBAL TEMPORARY TABLE statements, ensure that the work file database exists and that it has at least one table space with a 32 KB page size to avoid errors.

Adjust monitor programs that access OP buffers (change introduced in Version 9.1)

Adjust assignment strategies of monitor programs that access OP buffers. In Version 8, traces are left in a disabled state, which consumes CPU for trace data that cannot be retrieved. In Version 10, traces that are started with a destination of OPX choose the next available buffer that is not in use and traces are no longer left in a disabled state.

In addition, in Version 8, when the thread that owns an OP buffer terminates, OP traces are left in a disabled state and can be reactivated by starting another trace to that buffer. In Version 10, if an OP buffer terminates and the only destinations for the trace records are OP buffers, the traces that are started to that buffer are stopped. If an OP buffer terminates and the trace is started to both OP and non-OP destinations, the traces that are started to that buffer are modified to use non-OP destinations only.

The message format of DSNW128I and DSNW129I has changed, so modify automation that is based on those message formats.

Changed behavior for system-required objects (change introduced in Version 9.1)

After you migrate to Version 10 new-function mode, if the containing table space is implicitly created, you cannot drop any system-required objects, except for the LOB table space. This restriction applies even if you explicitly created these objects in a previous release. The following statements will not work properly if the system-required objects were implicitly created by DB2:

CREATE AUXILIARY TABLE

If you issue a CREATE AUXILIARY TABLE statement and an auxiliary table that was implicitly created by DB2 already exists for the same base table, the CREATE AUXILIARY TABLE statement fails and DB2 issues SQLCODE -646, SQLSTATE 55017, and reason code 3.

CREATE LOB TABLESPACE

If you issue a CREATE LOB TABLESPACE statement to create a LOB table space in an implicitly created database, the CREATE LOB TABLESPACE statement fails and DB2 issues SQLCODE -20355, SQLSTATE 429BW, and reason code 1.

CREATE DATABASE

If you specify an eight-character database name that begins with DSN and is followed by exactly five digits in a CREATE DATABASE statement, the CREATE DATABASE statement fails and DB2 issues SQLCODE -20074, SQLSTATE 42939.

CREATE INDEX

If you create an index on a primary key, unique key, or ROWID column that is defined as GENERATED BY DEFAULT, the index will be treated as a regular index instead of an enforcing index.

CREATE AUXILIARY INDEX

If you issue a CREATE AUXILIARY INDEX statement and an auxiliary index that was implicitly created by DB2 already exists for the same base table, the CREATE AUXILIARY INDEX statement fails and DB2 issues SQLCODE -748, SQLCODE 54048, and reason code 3.

CREATE

If you issue a CREATE statement and do not specify an IN clause or table space name, and the default buffer pool is not large enough, DB2 chooses a 4 KB, 8 KB, 16 KB, or 32 KB buffer pool, depending on the record size. If you issue a CREATE statement and do not specify an IN clause or table space name, DB2 implicitly creates a partitioned-by-growth universal table space. If you drop the table, DB2 also drops the containing table space.

DROP TABLE

If you issue a DROP TABLE statement to drop an auxiliary table from a table space that was implicitly created by DB2, the DROP TABLE statement fails and DB2 issues SQLCODE -20355, SQLSTATE 429BW, and reason code 2.

DROP TABLESPACE

If you issue a DROP TABLESPACE statement to drop an implicitly created LOB table space, the DROP TABLESPACE statement fails and DB2 issues SQLCODE -20355, SQLSTATE 429BW, and reason code 2.

DROP INDEX

If you issue a DROP INDEX statement to drop an enforcing primary key, unique key, or ROWID index from a table space that was implicitly created, the DROP INDEX statement fails and DB2 issues SQLCODE -669,

SQLSTATE 42917, and reason code 2. If you issue a DROP INDEX statement to drop an auxiliary index from a table space that was implicitly created, the DROP INDEX statement fails and DB2 issues SQLCODE -20355, SQLSTATE 429BW, and reason code 2.

Changes to INSERT, UPDATE, or DELETE statements on some indexes (change introduced in Version 9.1)

In Version 10, you cannot execute INSERT, UPDATE, or DELETE statements that affect an index in the same commit scope as ALTER INDEX statements on that index.

LOBs with a maximum length greater than 1 GB can now be logged (change introduced in Version 9.1)

In releases before Version 9.1, only LOBs with a maximum length of 1 GB or less could be logged. In Version 10, LOBs with a maximum length that is greater than 1 GB can be logged.

DB2 returns an error when a LOB value is specified for an argument to a stored procedure and the argument value is longer than the target parameter and the excess is not trailing blanks (change introduced in Version 9.1)

In releases before Version 9.1, DB2 did not return an error when a LOB value was specified for an argument to a stored procedure and the argument value was longer than the target parameter and the excess was not trailing blanks. DB2 truncated the data and the procedure executed. In Version 10, DB2 returns an error.

Changes to string formatting of decimal data

Explanation

DB2 Version 10 changed the formatting of decimal data by the CHAR and VARCHAR built-in functions and CAST specifications with a CHAR or VARCHAR result type. For input data that contains decimals, leading zeros are removed, and leading zeros are not added to values that did not already contain leading zeros. If the scale of the decimal value is zero, the decimal character is not returned. Also, the CHAR function no longer returns leading blanks for positive decimal values. The result of the CHAR function for decimal data is now consistent with the result of `CAST(decimal-expression AS CHAR)`.

After migration to Version 10, packages that were bound before Version 10 use the old behavior for these functions. Materialized query tables and indexes on expressions that were created before Version 10 also continue to use the old behavior.

Views and inline SQL functions use the behavior of the SQL statement that references the object. It is possible for references to the same view or function in different applications to get different behavior for these functions or casts.

Possible impact to your DB2 environment

These changes might cause unexpected output from applications that use the CHAR or VARCHAR functions for decimal data or the `CAST(decimal-expression AS CHAR)` or `CAST(decimal-expression AS VARCHAR)` specifications.

Actions to take

These changes occur in Version 10 conversion mode (from both Version 8 and Version 9.1). You can temporarily override these changes on a subsystem level by using the BIF_COMPATIBILITY subsystem parameter. You can also temporarily override these changes on an application level by adding schema SYSCOMPAT_V9 to the front of the PATH bind option or CURRENT PATH special register. This approach works for CHAR and VARCHAR functions and does not affect CAST specifications. The recommended approach is to modify your applications to handle the Version 10 behavior for these functions, as described in the following steps.

To modify your applications to handle the Version 10 behavior for CHAR, VARCHAR, and CAST:

1. Identify applications that need to be modified to handle this change. You can use IFCID trace 0366 to identify affected applications.
2. Ensure that the BIF_COMPATIBILITY subsystem parameter is set to V9_DECIMAL_VARCHAR.
To handle the change for the CHAR function only, you can set the subsystem parameter to V9 and complete the following steps for the CHAR function.
3. Change any affected applications to handle the new Version 10 CHAR and VARCHAR behavior, including stored procedures, non-inline user-defined functions, and trigger packages. Rewrite affected CAST specifications with the appropriate CHAR or VARCHAR function and a CAST to the correct length if needed.
4. Rebind and prepare packages with the PATH(SYSCURRENT,SYSSIBM) rebind option to use the new Version 10 CHAR and VARCHAR built-in functions. Repeat this step for native stored procedures (SQLPL) and non-inline SQL scalar functions.
5. For views that reference these casts or built-in functions, determine whether the view needs to be changed to have the expected output. Drop and re-create the views with the PATH(SYSCURRENT,SYSSIBM) rebind option only if necessary. Rebind any applications that reference the views with the PATH(SYSCURRENT,SYSSIBM) option to use the new Version 10 CHAR and VARCHAR built-in functions. Repeat this step for inline SQL scalar functions.
6. For materialized query tables or indexes on expressions that reference these casts or built-in functions, drop and re-create the materialized query tables or indexes on expressions with the PATH(SYSCURRENT,SYSSIBM) rebind option. Issue the REFRESH TABLE statement for materialized query tables. Rebind any applications that reference the materialized query tables or indexes on expressions with the PATH(SYSCURRENT,SYSSIBM) option to use the new Version 10 CHAR and VARCHAR built-in functions.
7. Change the value of the BIF_COMPATIBILITY subsystem parameter to CURRENT. When the subsystem parameter value is CURRENT, new applications, rebinds, and CREATE statements use the new CHAR, VARCHAR, and CAST behavior.

Materialized query tables and indexes on expressions use the CHAR, VARCHAR, and CAST behavior that is specified during its creation. If a reference statement has a different behavior that is specified by the BIF_COMPATIBILITY parameter or a different path, the materialized query table or expression-based index is not used.

Related reference:

☞ BIF COMPATIBILITY field (BIF_COMPATIBILITY subsystem parameter) (DB2 Installation and Migration)

Change to maximum result length of VARCHAR function

In Version 10, the maximum result length of the VARCHAR function is changed from 32767 to the maximum length of a VARCHAR.

Leading or trailing blanks from the VARCHAR_FORMAT function format string are no longer removed (change introduced in Version 9.1)

Leading or trailing blanks from the format string for the VARCHAR_FORMAT function are no longer removed. Existing view definitions are recalculated as part of Version 10, so the new rules take effect. You can continue to use existing SQL statements that use a materialized query table that references the VARCHAR_FORMAT function, but they use the old rules and remove leading and trailing blanks. Existing references to the VARCHAR_FORMAT function in bound statements only get the new behavior when they have been bound or rebound in Version 10.

Changes to VARCHAR_FORMAT function length attribute

In Version 10, for VARCHAR_FORMAT functions, the length attribute of the result is the length attribute of the format string, up to a maximum of 255. To apply this change, use the following guidance:

- Existing view definitions that reference the VARCHAR_FORMAT function should be regenerated with an ALTER VIEW statement.
- Existing materialized query statements that reference the VARCHAR_FORMAT function should be dropped and re-created.
- Bound SQL statements that reference the VARCHAR_FORMAT function will only use the new behavior when they have been bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- DESCRIBE statements will only determine the result data type for the VARCHAR_FORMAT function using the modified rules in the DESCRIBE statements have been bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- Existing indexes that involve an expression that reference the VARCHAR_FORMAT function should be dropped and re-created.

If an application is using the DSNTIAUL program, the result string is padded with characters '00'X. Consider this incompatible change for any applications that use the VARCHAR_FORMAT function and are dependent on the output from DSNTIAUL.

New format element for VARCHAR_FORMAT function

In Version 10, a new format element, “RRRR”, is supported. In previous versions, this format element was interpreted as two adjacent specifications of the “RR” format element.

Change to result of VARCHAR_FORMAT function with “HH12” format element

In Version 10, if the “HH12” format element is specified in a VARCHAR_FORMAT function and the time component of the first argument is 24:00:00, the input

timestamp value is adjusted to 00:00:00 and the date is changed to the next day. In previous releases, the timestamp value is adjusted to 12:00:00 and the day is not changed.

Change to result of VARCHAR_FORMAT function with “J” format element

In Version 10, if the “J” format element is specified in a VARCHAR_FORMAT function, the result is different from the result of the JULIAN_DAY function for dates earlier than October 15, 1582.

New supported data types for VARCHAR_FORMAT function

The VARCHAR_FORMAT function has been extended to allow date, character, and graphic string input for the first argument, and graphic string input for the second argument. If the function is invoked with one of the newly supported data types, and an existing user-defined function named VARCHAR_FORMAT also supports the data type, the function might resolve to the built-in function rather than the user-defined function. If the reference to the existing function uses the unqualified name and SYSIBM precedes the schema that was used for the user-defined function, the new function will be invoked rather than the user-defined function.

Change to maximum result length of REPEAT function

In Version 10, the maximum result length of the REPEAT function is changed from 32767 to the maximum length of a VARCHAR.

Change to maximum result length of XMLTABLE function

In Version 10, the maximum length of a VARCHAR data type result column of the XMLTABLE function is changed from 32767 to the maximum length of a VARCHAR.

DB2 drops certain indexes when a unique constraint is dropped (change introduced in Version 9.1)

In releases before Version 9.1, if a unique constraint was dropped, DB2 did not drop the index that enforced uniqueness. Starting in Version 9, if a table is in an implicitly created table space, and a unique constraint on that table is dropped, DB2 drops the index that enforces uniqueness.

DB2 enforces restriction on specifying a CAST FROM clause for some forms of CREATE FUNCTION statements (change introduced in Version 9.1)

The CAST FROM clause is included only in the syntax diagram for the CREATE FUNCTION statement for an external scalar function. The CAST FROM clause is not included in the syntax diagrams for the other variations of CREATE FUNCTION (external table function, sourced function, or SQL function); the clause cannot be used for these other variations. In previous releases, if you specified a CAST FROM clause in an unsupported context, you received no errors. Starting in Version 9, if a CAST FROM clause is specified in an unsupported context, DB2 issues an error.

DB2 enforces restrictions on specifying the AS LOCATOR clause and TABLE LIKE clause (change introduced in Version 9.1)

The AS LOCATOR clause for LOBs is included in the syntax diagram for the CREATE FUNCTION statement for an SQL function. This clause is not supported in other contexts when identifying an existing SQL function such as in an ALTER, COMMENT, DROP, GRANT, or REVOKE statement. In previous releases, if you specified an AS LOCATOR clause for LOBs in an unsupported context, you might not have received an error. Starting in Version 9, if an AS LOCATOR clause for LOBs is specified in an unsupported context, DB2 issues an error.

The TABLE LIKE clause for a trigger transition table is included only in the syntax diagram for the CREATE FUNCTION statement for an external scalar function, external table function, or sourced function. This clause is not supported for SQL functions or in other contexts when identifying an existing function such as in an ALTER, COMMENT, DROP, GRANT, or REVOKE statement, or in the SOURCE clause of a CREATE FUNCTION statement. In previous releases, if you specified a TABLE LIKE clause for a trigger transition table in an unsupported context, you might not have received an error. Starting in Version 9, if a TABLE LIKE clause for a trigger transition table is specified in an unsupported context, DB2 issues an error.

DB2 enforces restriction on the CCSID parameter for the DECRYPT_BIT and DECRYPT_BINARY functions (change introduced in Version 9.1)

The CCSID parameter is not supported by the DECRYPT_BIT and DECRYPT_BINARY built-in functions. In previous releases, if you specified an argument for the CCSID parameter for these functions, you received no errors. Starting in Version 9, if an argument is specified for the CCSID parameter in an unsupported context, DB2 issues an error.

Changed behavior of CREATE PROCEDURE for an SQL procedure (change introduced in Version 9.1)

With the introduction of native SQL procedures in Version 9, the semantics of the CREATE PROCEDURE statement for an SQL procedure has changed. Starting in Version 9, all SQL procedures that are created without the FENCED option or the EXTERNAL option in the CREATE PROCEDURE statement are native SQL procedures. In previous releases of DB2, if you did not specify either of these options, the procedures were created as external SQL procedures.

If you do specify FENCED or EXTERNAL, the meanings are the same as in previous releases of DB2. Both of these keywords mean that an external SQL procedure is to be created.

Explicitly qualify names of variables, parameters, and columns in SQL procedures (change introduced in Version 9.1)

As of Version 9, the rules that are used for name resolution within a native SQL procedure differ from the rules that were used for SQL procedures in prior releases. Because an SQL parameter or SQL variable can have the same name as a column name, you should explicitly qualify the names of any SQL parameters, SQL variables or columns that have non-unique names. For more information

about how the names of these items are resolved, see References to SQL parameters and SQL variables (DB2 SQL). The rules that are used for name resolution within external SQL procedures remain unchanged.

Make any necessary program changes for possibly different values for RETURNED_SQLSTATE and DB2_RETURNED_SQLCODE (change introduced in Version 9.1)

Starting in Version 9, when an SQL statement other than GET DIAGNOSTICS or compound-statement is processed, the current diagnostics area is cleared before DB2 processes the SQL statement. Clearing of the diagnostics area can result in different values being returned for RETURNED_SQLSTATE and DB2_RETURNED_SQLCODE for a GET DIAGNOSTICS statement than what would be returned if the GET DIAGNOSTICS statement were issued from within an external SQL procedure. Additionally, there might be some differences in the values returned for the SQLSTATE and SQLCODE SQL variables than would have been returned from an external SQL procedure. (External SQL procedures were previously called SQL procedures in Version 8.)

SQLSTATE and SQLCODE SQL variables after a GET DIAGNOSTICS statement (change introduced in Version 9.1)

Starting in Version 9, the SQLSTATE and SQLCODE SQL variables are not cleared following a GET DIAGNOSTICS statement.

Coding multiple SQL statements in a handler body (change introduced in Version 9.1)

Previous releases of DB2 did not allow for a compound statement within a handler. A workaround to include multiple statements within a handler (without support for a compound statement in a handler) was to use another control statement, such as an IF statement, which in turn contained multiple statements. Starting in Version 9, DB2 supports a compound statement within a handler body. The compound statement is recommended for including multiple statements within a handler body.

Unhandled warnings (change introduced in Version 9.1)

Starting in Version 9, when a native SQL procedure completes processing with an unhandled warning, DB2 returns the unhandled warning to the calling application. The behavior of an external SQL procedure is unchanged from releases prior to Version 9. When such a procedure completes processing with an unhandled warning, DB2 does not return the unhandled warning to the calling application.

Change your programs to handle any changed messages from SQL procedures (change introduced in Version 9.1)

Starting in Version 9, DB2 issues different messages for the new native SQL procedures than it does for external SQL procedures. (External SQL procedures were previously called SQL procedures in Version 8.) For external SQL procedures, DB2 continues to issue DSNHxxxx messages. For native SQL procedures, DB2 issues SQL return codes. The relationship between these messages is shown in the following table:

Table 1. Relationship between DSNHxxxx messages that are issued for external SQL procedures and SQLCODEs that are issued for native SQL procedures

DSNHxxxx message ¹	SQLCODE ²
DSNH051I	-051
DSNH385I	+385
DSNH590I	-590
DSNH4408I	-408
DSNH4777I	n/a
DSNH4778I	-778
DSNH4779I	-779
DSNH4780I	-780
DSNH4781I	-781
DSNH4782I	-782
DSNH4785I	-785
DSNH4787I	-787

Note:

1. These messages are used for external SQL procedures, which can be defined by specifying EXTERNAL or FENCED in Version 10.
2. These messages are used for native SQL procedures in Version 10.

Enhanced data type checking for zero-length characters (change introduced in Version 9.1)

Starting in Version 9, when you specify a CHAR data type with a length of 0 in the SQLDA, DB2 issues SQLCODE -804 regardless of the null indicator value.

Adding a column generates a new table space version (change introduced in Version 9.1)

In previous releases, adding a column to a table did not generate a new table space version. Starting in Version 9, adding a column to a table with an ALTER TABLE ADD COLUMN statement generates a new table space version.

You cannot add a column and issue SELECT, INSERT, UPDATE, or DELETE statements in the same commit scope

You cannot have a version-generating ALTER TABLE ADD COLUMN statement and SELECT, INSERT, UPDATE, or DELETE statements in the same commit scope. If a version-generating ALTER TABLE ADD COLUMN statement follows SELECT, INSERT, UPDATE, or DELETE statements in the same commit scope, SQLCODE -910 is issued. SQLCODE -910 is also issued if SELECT, INSERT, UPDATE, or DELETE statements follow a version-generating ALTER TABLE ADD COLUMN statement in the same commit scope.

CAST FROM clause of CREATE FUNCTION statement for SQL functions is no longer supported (change introduced in Version 9.1)

The CAST FROM clause of the CREATE FUNCTION statement for SQL functions is no longer supported. Starting in Version 9, if you issue a CREATE FUNCTION statement for an SQL function with a CAST FROM clause, DB2 issues an error.

Specifying ALTER DATABASE STOGROUP for work file databases (change introduced in Version 9.1)

In previous releases of DB2, you could not execute ALTER DATABASE STOGROUP on a work file database. Beginning with DB2 Version 9.1 conversion mode, this restriction is removed.

DB2 enforces restrictions about where an INTO clause can be specified (change introduced in Version 9.1)

The INTO clause (as related to queries) is included only in the syntax diagram for the SELECT INTO statement. The INTO clause is not included in the syntax diagrams for select-clause, subselect, fullselect, or select-statement. In previous releases, if you specified an INTO clause in an unsupported context in a query, you might not have received an error. Starting in Version 9, if an INTO clause is specified in an unsupported context, DB2 issues an error.

Change to how a positive, signed integer in an ORDER BY clause is treated

Explanation

Beginning in Version 10 conversion mode (from both Version 8 and Version 9.1), a positive, signed integer in an ORDER BY clause is treated as a *sort-key-expression*. Such integers were previously interpreted as column numbers.

For example, in previous versions of DB2, ORDER BY +1 in the following SELECT statement meant order by column 1 (C1).

```
SELECT C1, C2 FROM T1 ORDER BY +1;
```

Starting in Version 10, +1 means the constant +1, which has no effect on the order of the rows.

Possible impact to your DB2 environment

This change might cause unexpected results of queries that contain an ORDER BY clause with a positive, signed integer. However, no error is issued when such queries are run.

Actions to take

To prepare for this change, identify any queries that use a positive, signed integer in an ORDER BY clause to refer to a column in the result table. Modify these queries to use unsigned integers to identify column numbers.

Related reference:

 [order-by-clause \(DB2 SQL\)](#)

Binding DBRMs directly into plans is no longer supported

For pre-existing plans that are bound from DBRMs, you can use the COLLID parameter of the REBIND PLAN command to create packages. If you execute a plan that is bound from DBRMs, DB2 performs an automatic rebind that creates packages from the DBRMs and binds those packages into a plan. However, the recommendation is to use REBIND with the COLLID option so that you can specify bind options and receive more diagnostic information. If the installation uses the RACF access control module, owners of plans with DBRMs need to explicitly rebind the plans to convert the DBRMs to packages.

Some BIND PLAN and REBIND PLAN command options are no longer supported

The ACQUIRE(ALLOCATE) option of the BIND PLAN and REBIND PLAN commands is no longer supported. If you specify ACQUIRE(ALLOCATE), DB2 issues a warning message and uses ACQUIRE(USE).

Also, the MEMBER option of BIND PLAN and REBIND PLAN is no longer supported. If you specify MEMBER, DB2 issues a warning message, binds the specified DBRM into a package, and binds the package into a plan.

Plans and packages should be converted to DRDA protocol

Plans and packages that were previously bound using DBPROTOCOL(PRIVATE) should be converted to DRDA[®] protocol before migration to Version 10. In Version 10, plans and packages that were bound with the DBPROTOCOL(PRIVATE) bind option and access remote locations cannot run. Applications that use packages or plans that were bound with DBPROTOCOL(PRIVATE) and access remote locations fail with SQLCODE -904. A rebind of those plans and packages must be explicitly performed before they can execute successfully. Job DSNTIJPM identifies the objects that must be converted to use DRDA protocol.

Change to GRANT statement

The PUBLIC AT ALL LOCATIONS clause is no longer allowed in the GRANT statement for table and view privileges as an alternative to PUBLIC. The DSNTPPCK program in Version 8 and Version 9.1 analyzes the embedded SQL statements in packages and plans for private protocol SQL, which is invalid in Version 10. The program produces a report that states which packages and member DBRMs of plans contain the invalid syntax. The program scans either the SYSIBM.SYSSTMT catalog table, the SYSIBM.SYSPACKSTMT catalog table, or both. Applications that issue dynamic SQL statements with the invalid PUBLIC AT ALL LOCATIONS clause will receive SQLCODE -199.

Change to IMMEDIATE option of BIND PACKAGE command

In DB2 Version 10, if IMMEDIATE is not specified on a BIND PACKAGE command, the default is "I", or INHERITFROMPLAN. In previous versions, the default was NO.

Changes to conversion of special characters in collection IDs and package names

In Version 10, DRDA character type parameter data is sent between client drivers and DB2 in UTF-8 Unicode, if those client drivers also have this support. Character type parameter data includes package names and collection IDs. Prior to Version 10, collection IDs and package names were sent in EBCDIC, and then converted to

Unicode before being stored in the DB2 catalog. A collection ID or package name that is in the DB2 catalog from a bind that was initiated by an older driver might not match an ID or name that is sent to DB2 by a newer driver. This mismatch, which is caused by the way that some special characters are converted, can cause package-not-found errors. Job step DSNTGEN of job DSNTIJPB identifies package names and collection IDs that contain special characters that cause mismatches. From the upgraded client drivers, those packages need to be bound with the ACTION(REPLACE) option.

Changes to the RELEASE bind option

In releases prior to Version 10, the RELEASE bind option had no effect on database access threads. Starting in Version 10, by default, DB2 honors the RELEASE bind option for database access threads. You can modify this behavior by using the new MODIFY DDF PKGREL command.

Database metadata stored procedures are converted to Unicode

In DB2 Version 10, DB2-supplied database metadata stored procedures are encoded in Unicode. The Version 10 migration process redefines the stored procedures to use the following new load modules:

Table 2. Metadata stored procedures and associated load modules

Stored procedure	Load module
SYSIBM.SQLCOLPRIVILEGES	DSNACPRU
SYSIBM.SQLCOLUMNS	DSNACOLU
SYSIBM.SQLFOREIGNKEYS	DSNAFNKU
SYSIBM.SQLFUNCTIONCOLS	DSNAFCOU
SYSIBM.SQLFUNCTIONS	DSNAFUNU
SYSIBM.SQLGETTYPEINFO	DSNATYPU
SYSIBM.SQLPRIMARYKEYS	DSNAPRKU
SYSIBM.SQLPROCEDURECOLS	DSNAPCOU
SYSIBM.SQLPROCEDURES	DSNAPRCU
SYSIBM.SQLSPECIALCOLUMNS	DSNASPCU
SYSIBM.SQLSTATISTICS	DSNASTAU
SYSIBM.SQLTABLEPRIVILEGES	DSNATBPU
SYSIBM.SQLTABLES	DSNATBLU
SYSIBM.SQLUUDTS	DSNAUDTU

Some applications call the database metadata stored procedures to retrieve double-byte (DBCS) data. Those applications must be modified to use Unicode if they are bound on a DB2 server that has an EBCDIC SBCS CCSID, and the MIXED parameter is set to NO in the application defaults load module, *dsnhdccp*.

AUTHID is the default owner of packages that are bound by DSNTTRIN

In previous releases, the stored procedures and user-defined functions that are provided as part of the DB2 base product (DB2-supplied routines) were created and bound by processing inline DDL and bind statements in DSNTIJSB and other

| installation jobs. By default, the ID that was used to run the job was also the
| authorization ID for creating the routines and the default package owner for those
| routines.

| In Version 10, DB2-supplied routines are created and bound by running program
| DSNTRIN in job DSNTIJRT. The AUTHID parameter of DSNTRIN specifies the
| authorization ID for creating the routines. This ID is also the default owner of
| packages that are bound for those routines.

New default DEFINE attribute for dependent objects

As of Version 10, if the DEFINE attribute is not specified in the CREATE statement for explicitly created dependent objects (auxiliary indexes, XML indexes, and base table indexes), DB2 uses the DEFINE attribute of the base table space.

Exception: If the DEFINE attribute is not specified for explicitly created LOB table spaces, there is no correlation with the base table space until the auxiliary table is created. The DEFINE attribute is not inherited from the base, and the default is DEFINE YES.

Implicitly created dependent objects (base table indexes, LOB and XML table spaces, and their dependent indexes) inherit the DEFINE attribute of the base table space if it is DEFINE NO. Otherwise, if the base table space attribute is DEFINE YES, the value of the IMPDSEDF subsystem parameter is used for the dependent objects.

Change for creating partitioned table spaces

| To create a partitioned (non-universal) table space in DB2 Version 10 new-function
| mode, you must specify SEGSIZE 0 and the Numparts keyword of the CREATE
| TABLESPACE statement. Before new-function mode, do not specify the SEGSIZE to
| create a partitioned table space.

Change to default for CREATE TABLESPACE statements

| In previous releases of DB2, if a CREATE TABLESPACE statement contains the
| Numparts clause but neither the MAXPARTITIONS clause nor the SEGSIZE
| clause, a partitioned (non-universal) table space is created. Beginning in Version 10
| new-function mode, the same statement results in a range-partitioned table space
| with a segment size of 32 by default. You might observe this difference when a
| subsequent CREATE INDEX statement with the specified ranges fails because
| index partitioning of a range-partitioned table space is not supported.

| In response, you can change the default segment size through the
| DSN6SYSP.DPSEGSZ subsystem parameter, which is externalized as the DEFAULT
| PARTITION SEGSIZE field on panel DSNTIP7. The value of the DPSEGSZ
| parameter can range from 0 to 64 in increments of 4; for example, 0, 4, 8, and so on
| up to 64. The default is 32.

| When DPSEGSZ is set to 0, a CREATE TABLESPACE statement that contains the
| Numparts clause but neither the MAXPARTITIONS clause nor the SEGSIZE
| clause results in a partitioned (non-universal) table space, which is the behavior of
| previous releases. Note that the DPSEGSZ parameter is provided for compatibility
| only. It is deprecated in Version 10, and you should take steps to modify affected
| CREATE TABLESPACE statements or plan to use partition-by-growth table spaces
| instead of partitioned (non-universal) table spaces.

Upgrade to supported COBOL and PL/I compilers

If you use the Version 10 precompiler, you must upgrade to COBOL compilers that DB2 10 for z/OS supports. See the DB2 Program Directory for information about supported compilers. The generated SQLCA and SQLDA for COBOL have changed. The generated attribute for binary data items is now COMP-5 instead of COMP-4 or COMP. Applications that are compiled on compilers that do not support the COMP-5 attribute no longer work.

For some COBOL and PL/I compilers that are no longer supported, you can use a version of the precompiler that allows you to precompile applications that have dependencies on these unsupported compilers. You can use this version of the precompiler with the following unsupported compilers:

- OS/VS COBOL V1.2.4
- OS PL/I 1.5 (PL/I Opt. V1.5.1)
- VS/COBOL II V1R4
- OS PL/I 2.3

The load module for this precompiler is DSNHPC7. This precompiler is meant only to ease the transition from unsupported compilers to supported compilers. This precompiler has the following restrictions:

- There is no corresponding DB2 coprocessor function to match this precompiler.
- The precompiler does not support SQL procedures.
- Only COBOL and PL/I are supported.
- The SQL flagger is not supported.
- The precompiler produces Version 7 DBRMs, and does not support any capability that is newer than Version 7.
- The application defaults module must be named DSNHDECP.

Support of this precompiler is deprecated in Version 10.

GRAPHIC and NOGRAPHIC SQL processing options are removed

If you specify the SQL processing options GRAPHIC or NOGRAPHIC, DB2 issues a standard warning message about an invalid option. These options are superseded by the CCSID SQL processing option.

SELECT FROM data change statements in BEFORE triggers no longer supported

The following statements are no longer allowed in the body of a BEFORE trigger:

- SELECT FROM DELETE
- SELECT FROM INSERT
- SELECT FROM MERGE
- SELECT FROM UPDATE

RETURN statement in scalar functions must follow *option-list*

In previous DB2 versions, RETURN statements in scalar functions could be in any order, relative to other clauses. After migration to Version 10 conversion mode (from Version 8 or Version 9.1), the RETURN statement in CREATE FUNCTION or ALTER FUNCTION statements must follow *option-list*. If a RETURN statement precedes *option-list* in one of these statements, DB2 issues SQLCODE -199.

Changes to ROUND_TIMESTAMP and TRUNC_TIMESTAMP functions

In Version 10, the ROUND_TIMESTAMP and TRUNC_TIMESTAMP functions return the first day of the first ISO week of the ISO year when an ISO year format is specified.

Also, whenever a CC or SCC format is specified, a start of a century is considered to be year 01.

Changes to result of NEXT_DAY function

In Version 10, the result data type for the NEXT_DAY function is determined from the input data. If the first input argument is a string, the result is TIMESTAMP(6) WITHOUT TIME ZONE. Otherwise, the data type of the result is the same as the data type of the first input argument. For example, if the input is a date data type, the result is also a date data type. The following rules apply to the change in the NEXT_DAY function:

- Existing view definitions that reference the NEXT_DAY function should be regenerated with an ALTER VIEW statement.
- Existing materialized query statements that reference the NEXT_DAY function should be dropped and re-created.
- Bound SQL statements that reference the NEXT_DAY function only use the modified rules to determine the result data type if the statements are bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- DESCRIBE statements only use the modified rules to determine the result data type for the NEXT_DAY function if the statements are bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- Existing indexes that involve an expression that reference the NEXT_DAY function should be dropped and re-created.

Changes to MONTHS_BETWEEN function

In previous releases, the MONTHS_BETWEEN function ignored the time portion of arguments. In Version 10, the MONTHS_BETWEEN function takes the time portion of arguments into consideration.

Changes to TIMESTAMPDIFF function

In previous releases, the TIMESTAMPDIFF function allowed string input values that had more than six digits to the right of the decimal point. In Version 10, an error is issued if the string input value for TIMESTAMPDIFF function has more than six digits to the right of the decimal point.

Static SQL applications that use parallelism

DB2 incrementally rebinds the statements that use parallelism after migration to Version 10. Incremental rebinds can cause performance degradation. If you use the access control authorization exit (DSNX@XAC) for authorization, incremental rebinds can cause authorization failures because they trigger authorization checks of static SQL statements when the package is executed. The authorization checks are performed on the primary authorization ID during incremental rebinds.

You should manually rebind those statements that use parallelism after migration. You can run a query in job DSNTIJPB before you migrate to determine which statements can use parallelism, and are therefore candidates for incremental rebinds. You should consider rebinding those statements after migration, as soon as

your Version 10 system is stable. After you migrate to Version 10, you can also run a performance trace, class 3 or class 10 for IFCID 360, to identify the plans and packages that contain static SQL queries that use parallelism, and therefore need to be rebound.

Enforced SELECT authorization checking for UPDATE and DELETE statements

DB2 Version 10 checks for the SELECT privilege or appropriate administrative privilege before allowing a user to execute UPDATE or DELETE statements that reference an existing value in the target table. This authorization checking applies regardless of how the statements are executed (for example, bound in a package or executed in a dynamic statement). The authorization now also applies regardless of the setting of the SQLRULES(STD) bind option for static statements or the CURRENT RULES special register for dynamic statements. If the user does not have the necessary SELECT authorization or administrative privilege, a negative SQLCODE is returned.

Increased limit for work file record length

In Version 10 new-function mode, the limit for the row length in the result of a JOIN or the row length of a SORT record is increased from 32 767 bytes (1 page) to 65 529 bytes. The sort key maximum length is also increased from 16 000 bytes to 32 000 bytes. Applications that exceed the old limits fail in conversion mode (from both Version 8 and Version 9.1) with SQLCODE -670 or SQLCODE -136.

New restrictions for EXPLAIN tables

In DB2 Version 10 conversion mode (from both Version 8 and Version 9.1), EXPLAIN tables must be in Version 8 or later format and preferably encoded in Unicode. When EXPLAIN tables are in a format prior to the Version 8 format, DB2 returns SQLCODE -20008 reason code 2 for statements or commands that invoke EXPLAIN processing. Statements or commands that invoke EXPLAIN processing return SQLCODE +20520 reason code 2 if an EXPLAIN table is in Version 8 or Version 9.1 format, regardless of the encoding type. If an EXPLAIN table is in Version 10 format and encoded in EBCDIC, SQLCODE -878 is returned. When you convert EXPLAIN tables to Unicode encoding, applications that join with EXPLAIN tables might have different results because of the CCSID conversion. For more information, see Objects with different CCSIDs in the same SQL statement (DB2 Internationalization Guide).

Recommendation: Before you begin migration to Version 10, convert all EXPLAIN tables to the current version (8 or 9.1) format and Unicode encoding, and then check for joins to those tables. You can use job DSNTIJXA to convert most tables to current release format. You can use jobs DSNTIJXB and DSNTIJXC to migrate EBCDIC-encoded EXPLAIN tables to Unicode.

MEMBER CLUSTER table spaces indicated by MEMBER_CLUSTER column

In previous versions of DB2, a “K” or “I” in the TYPE column of the SYSTABLESPACE catalog table indicated that the table space had MEMBER CLUSTER structure. In Version 10, a new MEMBER_CLUSTER column on the SYSTABLESPACE catalog table is populated during the enabling-new-function mode (from both Version 8 and Version 9.1) migration process. For existing MEMBER CLUSTER table spaces, values of “K” in the TYPE column of SYSTABLESPACE are replaced with “L”, and values of “I” are replaced with blank.

The MEMBER_CLUSTER column is populated with "Y". After migration to enabling-new-function mode (from Version 8 or Version 9.1), applications that query "K" or "I" in the TYPE column must query the new MEMBER_CLUSTER column instead.

Changed values for the modification level in the product signature

The DB2 Version 10 product signature has the form DSN1001 m , where m is the modification level. Values 0 and 1 are reserved for maintenance levels in conversion mode from Version 8, conversion mode* from Version 8, enabling-new-function mode from Version 8, and enabling-new-function mode* from Version 8. Values 2 and 3 are for maintenance levels in conversion mode from Version 9.1, conversion mode* from Version 9.1, enabling-new-function mode from Version 9.1, and enabling-new-function mode* from Version 9.1. Values 5, 6, 7, 8, and 9 are for maintenance levels in new-function mode. Value 4 is undefined.

Changed behavior for the CREATE FUNCTION statement

In all forms of the CREATE FUNCTION statement, a parameter list is required. Functions without parameters must have empty parentheses specified, as in the following example: CREATE FUNCTION F1 () RETURNS INT RETURN 1.

In previous releases, if you specified CREATE FUNCTION without a parameter list (for example, CREATE FUNCTION F1 RETURNS INT RETURN 1), you received no errors. In Version 10, if you specify CREATE FUNCTION without a parameter list, DB2 issues an error.

Also, a CREATE FUNCTION statement for a non-inline SQL scalar function cannot have a parameter that is a distinct type that is based on a LOB data type. The inline SQL scalar functions have supported, and will continue to support, distinct type parameters where the underlying base data type is a LOB. For the syntax of this statement, see CREATE FUNCTION in the DB2 for z/OS SQL Reference Guide.

Different SQLSTATE returned for some DELETE or UPDATE statements

In previous releases of DB2, a SQLSTATE value of "00000" was returned for some DELETE or UPDATE statements that did not contain a WHERE clause, when SQLWARN flags were set. Those situations occurred during bind processing. As of DB2 Version 10, a SQLSTATE value of "01504" is returned in those situations. This change does not affect SQL statements that are executed on behalf of a DB2 for z/OS requester.

Changed default behavior of multiple-row inserts for ODBC z/OS applications

As of DB2 Version 10, the default behavior for multiple-row inserts is changed from non-atomic to atomic. To change the default behavior back to non-atomic, set keyword PARAMOPTATOMIC = 0 in the data source section of the ODBC initialization file.

Changes to ALTER TABLESPACE statement error codes

Before Version 10 conversion mode (from Version 8 or Version 9.1), if you execute an ALTER TABLESPACE MAXPARTITIONS statement on catalog objects, DB2

issues SQLCODE -644. After you migrate to Version 10 conversion mode (from Version 8 or Version 9.1), DB2 issues SQLCODE -607.

Also before Version 10 conversion mode (from Version 8 or Version 9.1), if you execute an ALTER TABLESPACE MAXPARTITIONS statement with a value of MAXPARTITIONS that is invalid with the page size and DSSIZE values, DB2 issues SQLCODE -4701. After you migrate to Version 10 conversion mode (from Version 8 or Version 9.1), DB2 issues SQLCODE -644.

Change to CREATE and ALTER statements

CREATE and ALTER statements for native SQL procedures no longer support the DEFAULT keyword for the SQL PATH option.

Change to DESCRIBE statement

In Version 10, when a DESCRIBE statement is used on a result table that includes a distinct type, it will now return information about the distinct type.

New restrictions on using DSNTIAUL

DSNTIAUL can no longer be used to process CREATE FUNCTION (SQL scalar) statements that would result in a package or CREATE TRIGGER statements. DSNTIAUL also cannot be used to process any other statement that contains *SQL-routine-body*. These statements are CREATE PROCEDURE (SQL external), CREATE PROCEDURE (SQL native), CREATE FUNCTION (SQL table), ALTER PROCEDURE (SQL native) with an ADD or REPLACE clause, and ALTER FUNCTION (SQL scalar) with an ADD or REPLACE clause.

Changes to SYSROUTINES

As of Version 10, the CREATEDTS column of SYSIBM.SYSROUTINES will always reflect the time that a CREATE statement was first issued for a routine. In previous releases of DB2, the CREATEDTS column might have contained different values if multiple versions of a routine were created.

Catalog restructured

In Version 10, the DB2 catalog is restructured to reduce lock contention. As a result, binding takes longer in Version 10 than it did in previous versions. Also, execution of the DECLARE GLOBAL TEMPORARY TABLE statement takes longer in Version 10.

Changed data type for an untyped parameter marker

In previous releases, an untyped parameter marker for a unary minus operator has an assumed DOUBLE data type. In DB2 Version 10, the assumed data type is DECFLOAT(34).

Changes to handling of special values Infinity, sNaN, and NaN

In previous releases, when DB2 returns a decimal floating-point (DECFLOAT) value for Infinity, NaN, or sNaN as a CHAR or VARCHAR string, the string is in mixed case. In DB2 Version 10, the values are returned in upper case as follows: INFINITY, NAN, or SNAN.

Changes for INSTEAD OF triggers

Changes are introduced for existing INSTEAD OF triggers that were defined on a view that has a ROWID column or a column that is based on an underlying column of any of the following types:

- A security label column.
- A row change timestamp column.
- A column that is defined with AS ROW BEGIN.
- A column that is defined with AS ROW END.
- A column that is defined with AS TRANSACTION START ID.

For such INSTEAD OF triggers, the following actions now fail with a negative SQLCODE:

- Rebinding the INSTEAD OF trigger package.
- Dropping and re-creating the INSTEAD OF trigger.
- Running the REPAIR DBD utility on a database that includes the INSTEAD OF trigger.
- Issuing ALTER TABLE ALTER COLUMN SET DATA type on a table that has a view on which the INSTEAD OF trigger is defined.

Change to positioned update or delete statements

Explanation

In DB2 Version 10, packages must be rebound if both of the following statements are true:

- The package contains static cursors that were bound in a previous version.
- The package contains dynamic UPDATE or DELETE statements that contain the WHERE CURRENT of clause. The WHERE CURRENT of clause indicates a positioned UPDATE or DELETE statement.

Possible impact to your DB2 environment

If any such packages are not rebound in Version 10, positioned UPDATE and DELETE statements fail with SQLCODE -20249 when they run against a down-level cursor.

Actions to take

After migration to Version 10 conversion mode (from Version 8 or Version 9.1), rebind packages that contain static cursors with positioned UPDATE or DELETE statements.

Change to stored procedure parameter values returned to non-Java clients

In previous releases, when a remote application calls a DB2 for z/OS stored procedure, the data types of the returned output data match the data types of the corresponding CALL statement arguments. Starting in Version 10 conversion mode (from both Version 8 and Version 9.1), the data types of the returned output data match the data types of the parameters in the stored procedure definition. This change can cause conversion failures for some applications that use non-Java client drivers such as .NET to call stored procedures on DB2. To prepare for this change, modify the CALL requests of your non-Java™ client applications to specify

argument types that conform to the semantics of the client driver. If you want to temporarily override the Version 10 behavior, you can set the DDF_COMPATIBILITY subsystem parameter to SP_PARMS_NJV.

The SP_PARMS_NJV option of the DDF_COMPATIBILITY subsystem parameter is deprecated. Although the option is supported in Version 10, it will be removed in a later release of DB2.

Change to results of JDBC method PreparedStatement.setTimestamp

TIMESTAMP WITH TIME ZONE is first supported in DB2 for z/OS Version 10 new-function mode. Before Version 10 new-function mode, if the value that is assigned to a column (the second parameter of PreparedStatement.setTimestamp) has the java.sql.Timestamp data type, and the column data type is not known, the IBM Data Server Driver for JDBC and SQLJ chooses TIMESTAMP as the target data type. However, starting with DB2 for z/OS Version 10 new-function mode, unless the value that is assigned to the column is 0001-01-01-00:00:00.000000 or 9999-12-31-23:59:59.999999, the driver chooses TIMESTAMP WITH TIME ZONE as the column data type. If the driver chooses the TIMESTAMP data type, and the column type is actually TIMESTAMP WITH TIME ZONE, the database manager sets the time zone in the target column using the value of the IMPLICIT_TIMEZONE DECP value. This value might differ from the value that is inserted prior to Version 10 new-function mode.

To produce the same results before and after new-function mode when PreparedStatement.setTimestamp is executed, specify a com.ibm.db2.jcc.DBTimestamp value as the second parameter.

Change in how DB2 returns stored procedure output parameter data to remote clients

When an application on a client system calls a stored procedure on a DB2 10 for z/OS server, DB2 now handles the output parameters differently. Previously, DB2 returned stored procedure output parameters that were formatted according to the SQL type of the corresponding argument in the CALL statement. DB2 10 now returns output parameters that are formatted according to the SQL type of the corresponding parameter in the stored procedure declaration. This new behavior provides improved performance at the server by avoiding unnecessary server data conversions. Also, this new behavior is consistent with the existing server behavior for the return of query and select output data and it is consistent with the behavior of other DB2 family servers. Therefore, it provides applications with a more consistent and predictable interface to DB2.

In general, for applications that conform to client standards, this change has no impact to the calling application. In some cases, however, application changes might be needed. To prepare for this change, examine your remote applications that call DB2 stored procedures. If necessary, modify the CALL statements in your remote applications to specify argument data types that match the data types of the parameters in the stored procedure definitions.

When the change in behavior occurs

In general, the new behavior occurs after migration to DB2 10 for z/OS conversion mode from Version 8 (CM8) or conversion mode from Version 9.1 (CM9). However, the following exceptions apply:

- For applications using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to access a DB2 10 data sharing group in CM8 or CM9, where the enableSysplexWLB client property is set to true:

In such a configuration, the change in behavior occurs when the data sharing group is migrated to DB2 10 new-function mode (NFM). As long as the data sharing group is in CM8 or CM9 to support coexistence, DB2 uses the old behavior. The old behavior is used because the IBM Data Server Driver for JDBC and SQLJ might be caching data descriptors for outputs from the server. When the enableSysplexWLB property is set to true, the old behavior is maintained to ensure that the cached client descriptors are valid regardless of which member of the data sharing group is accessed.

To prepare for this change in behavior while DB2 10 is in CM8 or CM9 to support coexistence, test your applications with the enableSysplexWLB client property set to false. Or, test the applications against a stand-alone DB2 10 server, if such a system is available for testing. The new DB2 10 behavior will be used for each of those applications even though the DB2 data sharing group is in CM8 or CM9.

- For applications using the IBM Data Server Driver for JDBC and SQLJ or non-Java clients (such as .NET), Version 9 or earlier:

In such a configuration, the change in behavior occurs, by default, after migration to DB2 10 for z/OS CM8 or CM9. However, the change in behavior can be temporarily deferred. To temporarily override the DB2 10 behavior for returning stored procedure output parameter data to the IBM Data Server Driver for JDBC and SQLJ or non-Java clients only (such as .NET), set the DDF_COMPATIBILITY subsystem parameter to SP_PARMS_NJV. The SP_PARMS_NJV option of the DDF_COMPATIBILITY subsystem parameter is deprecated. Although the option is supported in DB2 10, it will be removed in a later release of DB2. For more information about DDF_COMPATIBILITY, see Subsystem parameters that are not on installation panels (DB2 Installation and Migration).

Examples of application impacts

The following examples use the IBM Data Server Driver for JDBC and SQLJ and .NET. Other drivers might behave differently. Because DB2 10 no longer converts data types when returning stored procedure output parameter data, there might be similar impacts to applications for other data types that are not discussed in the examples. If the calling application specifies arguments that are different from the declared parameter types, then the mismatch between the argument and parameter data types is handled by the client system, according to the client's programming model and the client's capabilities. If the client performs data type conversions such as for numeric data, the result of the conversion performed by the client might differ from the result that was previously returned when the DB2 server performed the conversion.

Example 1

A stored procedure parameter is declared as an INTEGER data type, but the application CALL statement specifies a SMALLINT argument.

- Behavior before DB2 10: DB2 converts the INTEGER data and returns a 2-byte SMALLINT value, which maps to an Int16 .NET data type.
- New behavior in DB2 10: DB2 returns a 4-byte INTEGER value, which maps to an Int32 .NET data type.

Sample application impact (.NET): If the application uses an IBM Data Server .NET client, the stored procedure CALL statement now fails with an invalid

conversion error because the Int16 and Int32 .NET data types are not compatible with each other. The application must be changed to specify an Int32 argument in the CALL statement as required by the .NET programming model. Applications that are coded with an Int32 argument and call stored procedures that have an INTEGER parameter require no change.

The .NET programming model enforces strong data typing. Thus, even though the argument type and the parameter type are compatible SQL types, the .NET driver enforces strong data type checking according to the .NET semantics. Refer to the .NET documentation for information about .NET strong type checking.

Example 2

A stored procedure parameter is declared as a REAL data type, but the application CALL statement specifies a DOUBLE argument.

- Behavior before DB2 10: DB2 converts the output data to a DOUBLE value before returning it to the client.
- New behavior in DB2 10: DB2 returns the data to the client as a REAL value.

Sample application impact (Java): If an application that uses the IBM Data Server Driver for JDBC and SQLJ uses the registerOutParameter() method to register the output parameter as a DOUBLE value before calling a stored procedure that has a REAL parameter, the following behavior occurs if the Java application uses the getObject() method to retrieve the output value:

- Before Version DB2 10, DB2 converted the REAL value to a DOUBLE value and returned it to the client. The IBM Data Server Driver for JDBC and SQLJ returned the DOUBLE value directly to the application as the result of the getObject() method.
- Beginning in DB2 10, DB2 returns the REAL value to the client. The IBM Data Server Driver for JDBC and SQLJ converts the REAL value to a DOUBLE value and returns it to application as the result of the getObject() method. APAR IC80974 for LUW clients and APAR PM58951 (JCC 3.63.131) and PM58952 (JCC 4.13.136) for z/OS clients are required in the IBM Data Server Driver for JDBC and SQLJ to ensure that the conversion is completed by the getObject() method. If the APAR fix is not applied, the Java application gets a ClassCastException.

Example 3

A stored procedure parameter is declared as a TIMESTAMP data type, but the application CALL statement specifies a VARCHAR argument.

- Behavior before DB2 10: DB2 converts the fixed-length TIMESTAMP data and returns a 26-byte VARCHAR value, containing a TIMESTAMP value in ISO format (*yyyy-mm-dd-hh.mm.ss[.ffffff]*). The VARCHAR value maps to a String .NET data type.
- New behavior in DB2 10: DB2 returns a 26-byte fixed-length TIMESTAMP value, containing a TIMESTAMP value in ISO format. The TIMESTAMP value maps to a DateTime .NET data type.

Sample application impact (Java): If the application uses the IBM Data Server Driver for JDBC and SQLJ, then an application change might be required, depending on what method the Java application uses to retrieve the parameter data.

- Before DB2 10, the getString() method previously returned TIMESTAMP data in ISO format, while the getTimestamp() method returned TIMESTAMP data in Java format

- Beginning in DB2 10, both the `getString()` and `getTimestamp()` methods return the `TIMESTAMP` value in Java format (`yyyy-mm-dd hh:mm:ss[.fffffffff]`).

An application change might be required if the Java application uses the `getString()` method and the application depends on receiving `TIMESTAMP` values in ISO format. Applications that use the `getTimestamp()` method require no change.

Similar considerations apply for `TIME` parameters. Beginning in DB2 10, DB2 returns `TIME` parameters as `TIME` data values in ISO format (`hh.mm.ss`), regardless of the SQL type of the corresponding argument in the `CALL` statement. For Java applications, the `getString()` method now returns `TIME` data in Java format (`hh.mm.ss`). If a Java application uses the `getString()` method and requires `TIME` data in ISO format, the application must be examined for possible changes.

Changes to datetime built-in functions

Explanation

Many datetime functions allow arguments containing *string representations of datetime values*. Valid formats for those strings are described in the DB2 SQL Reference.

In DB2 Version 10 conversion mode (from Version 8 or Version 9.1), the following additional string formats are allowed in the specified limited contexts:

- A string value of seven characters representing a date is allowed as an argument to the `DATE` function only.
- A string value of 8, 13, or 14 characters representing a point in time is allowed as an argument to the `TIMESTAMP` function only.

Possible impact to your DB2 environment

After migration to Version 10, applications that provide a seven-character string argument to represent a date for built-in functions other than the `DATE` function will return an error. Applications that provide a string value of 8, 13, or 14 characters to represent a point in time as an argument for built-in functions other than the `TIMESTAMP` function also return an error.

Actions to take

To prepare for this change, use valid string formats that are described in *String representations of datetime values (DB2 SQL)* in arguments for all of your datetime functions.

SQLCODE change for subsequent CAF CONNECT attempts

Explanation

In previous releases, a call attachment facility (CAF) `CONNECT` request that is followed by another `CONNECT` request without an intervening `disconnect` results in a zero return code. In DB2 Version 10, if the second `CONNECT` request is for a different or unknown subsystem, group attachment, or subgroup attachment name, a -924 `SQLCODE` is returned. If the second `CONNECT` request is for the same subsystem, group attachment, or subgroup attachment name, a +361 `SQLCODE` is returned. A failed attempt to connect to the CAF does not change the current connection. Therefore, in both of these scenarios, the existing CAF connection persists.

Possible impact to your DB2 environment

Some of your applications might receive a -924 or +361 return code where a zero return code was previously returned.

Actions to take

Review your applications for subsequent CONNECT requests for CAF. Modify these applications to handle the new SQLCODEs or rewrite the applications to remove subsequent CONNECT requests.

Change to serialization of an empty XML element

Explanation

In DB2 Version 9, the XML2CLOB function was deprecated, and in DB2 Version 10, XML2CLOB is no longer supported. The replacement for XML2CLOB is explicit or implicit execution of XMLSERIALIZE. XMLSERIALIZE and XML2CLOB serialize empty XML elements in different ways. XML2CLOB serializes an empty XML element as a start-element tag followed by an end-element tag (for example, `<emptyElement></emptyElement>`). XMLSERIALIZE serializes an empty XML element as a start-element tag followed by an end-element tag (for example, `<emptyElement></emptyElement>`), or as an empty-element tag (for example, `<emptyElement/>`).

Possible impact to your DB2 environment

Applications that depend on serialization of an empty XML element as a start-element tag followed by an end-element tag might receive errors.

Actions to take

After you migrate to DB2 Version 10, you can set subsystem parameter XML_RESTRICT_EMPTY_TAG to YES. This setting causes DB2 to always serialize an empty XML element as a start-element tag followed by an end-element tag.

Data types of output arguments from a stored procedure call in a Java application

In DB2 Version 10 new-function mode, when a Java application that uses the IBM Data Server Driver for JDBC and SQLJ calls a stored procedure, the data types of stored procedure output arguments match the data types of the parameters in the stored procedure definition.

Explanation

Before DB2 Version 10, if a Java client called a DB2 for z/OS stored procedure, the data types of output arguments matched the data types of the corresponding CALL statement arguments. Starting in DB2 Version 10, the data types of the output arguments match the data types of the parameters in the stored procedure definition.

Possible impact to your DB2 environment

If the version of the IBM Data Server Driver for JDBC and SQLJ is lower than 3.63 or 4.13, a `java.lang.ClassCastException` might be thrown when an output argument value is retrieved.

Actions to take

Take one of the following actions:

- Upgrade the IBM Data Server Driver for JDBC and SQLJ to version 3.63 or 4.13, or later.
- Modify the data types in `CallableStatement.registerOutParameter` method calls to match the parameter data types in the stored procedure definitions. You can set application compatibility to V10R1 and run a trace for IFCID 0366 or 0376 to identify affected applications. Trace records for those applications have a QW0366FN field value of 8.

Change to IBM Data Server Driver for JDBC and SQLJ handling of `TIMESTAMP WITH TIME ZONE` data type

Before DB2 Version 10 new-function mode, the `TIMESTAMP WITH TIME ZONE` data type was not supported. If a Java client application passed a timestamp input value to a `TIMESTAMP` column, the IBM Data Server Driver for JDBC and SQLJ did not include the local time zone with the timestamp value. Starting with DB2 Version 10 new-function mode, the `TIMESTAMP WITH TIME ZONE` data type is supported. If a Java client application passes a timestamp input value to a `TIMESTAMP` column, the IBM Data Server Driver for JDBC and SQLJ constructs a timestamp input value that includes the local time zone. If the value that the driver sends to the server is out of supported range for the server, the application receives `SQLCODE -181`. You can temporarily prevent this error by including `IGNORE_TZ` in the settings for the `DDF_COMPATIBILITY` subsystem parameter.

Delimiters used for accessing tables on DB2 for Linux, UNIX, and Windows

Explanation

DB2 Version 10 resolves aliases prior to sending SQL statements to a remote site for applications that use system-directed access. During a remote package bind against the remote site, modified SQL statement text is bound on the remote system. The `DRDA_RESOLVE_ALIAS` subsystem parameter is provided in DB2 Version 8 and DB2 Version 9.1 to help verify applications that are affected by this change of behavior before you migrate to DB2 Version 10.

Possible impact to your DB2 environment

This change can impact applications that access a DB2 for Linux, UNIX, and Windows server if the SQL preprocessing option `QUOTE` or `QUOTESQL` is used. `QUOTE` or `QUOTESQL` specifies that a quotation mark (") is used as the string delimiter and an apostrophe (') is used for SQL identifiers in SQL statements. This option does not control how the COBOL compiler processes string delimiters within the application program statements. DB2 for Linux, UNIX, and Windows does not support statement strings that have been precompiled under the `QUOTE` or `QUOTESQL` option and returns a warning on the `BIND` command. DB2 for z/OS does use the precompiler option to govern which string delimiter to use for SQL identifiers when modifying the SQL text. This causes a bind or rebind to fail on DB2 for Linux, UNIX, and Windows in DB2 Version 10 when the `QUOTE` or `QUOTESQL` precompiler option is used to generate the DBRM that is the source of the remote bind package processing.

Actions to take

When accessing a remote table on a DB2 for Linux, UNIX, and Windows server using an alias, applications must be precompiled using the APOST or APOSTSQL option. Character string literals must be delimited by apostrophes and SQL identifiers must be delimited by quotation marks.

Qualify user-defined function names

If you use a user-defined function that has the same name as a built-in function that has been added to Version 10, ensure that you fully qualify the function name. If the function name is unqualified and "SYSIBM" precedes the schema that you used for this function in the SQL path, DB2 invokes one of the built-in functions.

For a list of built-in functions, including those that have been added in Version 10, see Functions (DB2 SQL).

SQLCODE changes

Some SQLCODE numbers and message text might have changed in DB2 Version 10. Also, the conditions under which some SQLCODEs are issued might have changed.

SQL reserved words

Version 10 has several new SQL reserved words. Refer to Reserved words (DB2 SQL) for the list of reserved words, and adjust your applications accordingly.



Application and SQL release incompatibilities for migration from Version 9.1

When you migrate from DB2 Version 9.1 to Version 10, be aware of the application and SQL release incompatibilities.



Plan for the following changes in Version 10 that might affect your migration.

Release incompatibilities that were changed or added since the first edition of this Version 10 publication are indicated by a vertical bar in the left margin. In other areas of this publication, a vertical bar in the margin indicates a change or addition that has occurred since the Version 9.1 release of this publication.

Automatic rebind of plans and packages created before DB2 Version 6

If you have plans and packages that were bound before DB2 Version 6 and you specified YES or COEXIST in the AUTO BIND field of panel DSNTIPO, DB2 Version 10 autobinds these packages. Thus, you might experience an execution delay the first time that such a plan is loaded. Also, DB2 might change the access path due to the autobind, potentially resulting in a more efficient access path.

If you specify NO in the AUTO BIND field of panel DSNTIPO, DB2 Version 10 returns SQLCODE -908, SQLSTATE 23510 for each attempt to use such a package or plan until it is rebound.

IBMREQD is no longer reliable as a release dependency mark

The IBMREQD field in DB2 catalog tables is no longer a reliable indicator for determining release dependencies. Use the RELCREATED or RELBOUND fields instead.

Changes to string formatting of decimal data

Explanation

DB2 Version 10 changed the formatting of decimal data by the CHAR and VARCHAR built-in functions and CAST specifications with a CHAR or VARCHAR result type. For input data that contains decimals, leading zeros are removed, and leading zeros are not added to values that did not already contain leading zeros. If the scale of the decimal value is zero, the decimal character is not returned. Also, the CHAR function no longer returns leading blanks for positive decimal values. The result of the CHAR function for decimal data is now consistent with the result of `CAST(decimal-expression AS CHAR)`.

After migration to Version 10, packages that were bound before Version 10 use the old behavior for these functions. Materialized query tables and indexes on expressions that were created before Version 10 also continue to use the old behavior.

Views and inline SQL functions use the behavior of the SQL statement that references the object. It is possible for references to the same view or function in different applications to get different behavior for these functions or casts.

Possible impact to your DB2 environment

These changes might cause unexpected output from applications that use the CHAR or VARCHAR functions for decimal data or the `CAST(decimal-expression AS CHAR)` or `CAST(decimal-expression AS VARCHAR)` specifications.

Actions to take

These changes occur in Version 10 conversion mode (from both Version 8 and Version 9.1). You can temporarily override these changes on a subsystem level by using the BIF_COMPATIBILITY subsystem parameter. You can also temporarily override these changes on an application level by adding schema SYSCOMPAT_V9 to the front of the PATH bind option or CURRENT PATH special register. This approach works for CHAR and VARCHAR functions and does not affect CAST specifications. The recommended approach is to modify your applications to handle the Version 10 behavior for these functions, as described in the following steps.

To modify your applications to handle the Version 10 behavior for CHAR, VARCHAR, and CAST:

1. Identify applications that need to be modified to handle this change. You can use IFCID trace 0366 to identify affected applications.
2. Ensure that the BIF_COMPATIBILITY subsystem parameter is set to V9_DECIMAL_VARCHAR.

To handle the change for the CHAR function only, you can set the subsystem parameter to V9 and complete the following steps for the CHAR function.

3. Change any affected applications to handle the new Version 10 CHAR and VARCHAR behavior, including stored procedures, non-inline user-defined functions, and trigger packages. Rewrite affected CAST specifications with the appropriate CHAR or VARCHAR function and a CAST to the correct length if needed.
4. Rebind and prepare packages with the PATH(SYSCURRENT, SYSIBM) rebind option to use the new Version 10 CHAR and VARCHAR built-in functions. Repeat this step for native stored procedures (SQLPL) and non-inline SQL scalar functions.
5. For views that reference these casts or built-in functions, determine whether the view needs to be changed to have the expected output. Drop and re-create the views with the PATH(SYSCURRENT, SYSIBM) rebind option only if necessary. Rebind any applications that reference the views with the PATH(SYSCURRENT, SYSIBM) option to use the new Version 10 CHAR and VARCHAR built-in functions. Repeat this step for inline SQL scalar functions.
6. For materialized query tables or indexes on expressions that reference these casts or built-in functions, drop and re-create the materialized query tables or indexes on expressions with the PATH(SYSCURRENT, SYSIBM) rebind option. Issue the REFRESH TABLE statement for materialized query tables. Rebind any applications that reference the materialized query tables or indexes on expressions with the PATH(SYSCURRENT, SYSIBM) option to use the new Version 10 CHAR and VARCHAR built-in functions.
7. Change the value of the BIF_COMPATIBILITY subsystem parameter to CURRENT. When the subsystem parameter value is CURRENT, new applications, rebinds, and CREATE statements use the new CHAR, VARCHAR, and CAST behavior.

Materialized query tables and indexes on expressions use the CHAR, VARCHAR, and CAST behavior that is specified during its creation. If a reference statement has a different behavior that is specified by the BIF_COMPATIBILITY parameter or a different path, the materialized query table or expression-based index is not used.

Related reference:

 [BIF COMPATIBILITY field \(BIF_COMPATIBILITY subsystem parameter\) \(DB2 Installation and Migration\)](#)

Change to maximum result length of VARCHAR function

In Version 10, the maximum result length of the VARCHAR function is changed from 32767 to the maximum length of a VARCHAR.

Changes to VARCHAR_FORMAT function length attribute

In Version 10, for VARCHAR_FORMAT functions, the length attribute of the result is the length attribute of the format string, up to a maximum of 255. To apply this change, use the following guidance:

- Existing view definitions that reference the VARCHAR_FORMAT function should be regenerated with an ALTER VIEW statement.
- Existing materialized query statements that reference the VARCHAR_FORMAT function should be dropped and re-created.
- Bound SQL statements that reference the VARCHAR_FORMAT function will only use the new behavior when they have been bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.

- DESCRIBE statements will only determine the result data type for the VARCHAR_FORMAT function using the modified rules in the DESCRIBE statements have been bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- Existing indexes that involve an expression that reference the VARCHAR_FORMAT function should be dropped and re-created.

If an application is using the DSNTIAUL program, the result string is padded with characters '00'X. Consider this incompatible change for any applications that use the VARCHAR_FORMAT function and are dependent on the output from DSNTIAUL.

New format element for VARCHAR_FORMAT function

In Version 10, a new format element, “RRRR”, is supported. In previous versions, this format element was interpreted as two adjacent specifications of the “RR” format element.

Change to result of VARCHAR_FORMAT function with “HH12” format element

In Version 10, if the “HH12” format element is specified in a VARCHAR_FORMAT function and the time component of the first argument is 24:00:00, the input timestamp value is adjusted to 00:00:00 and the date is changed to the next day. In previous releases, the timestamp value is adjusted to 12:00:00 and the day is not changed.

Change to result of VARCHAR_FORMAT function with “J” format element

In Version 10, if the “J” format element is specified in a VARCHAR_FORMAT function, the result is different from the result of the JULIAN_DAY function for dates earlier than October 15, 1582.

New supported data types for VARCHAR_FORMAT function

The VARCHAR_FORMAT function has been extended to allow date, character, and graphic string input for the first argument, and graphic string input for the second argument. If the function is invoked with one of the newly supported data types, and an existing user-defined function named VARCHAR_FORMAT also supports the data type, the function might resolve to the built-in function rather than the user-defined function. If the reference to the existing function uses the unqualified name and SYSIBM precedes the schema that was used for the user-defined function, the new function will be invoked rather than the user-defined function.

Change to maximum result length of REPEAT function

In Version 10, the maximum result length of the REPEAT function is changed from 32767 to the maximum length of a VARCHAR.

Change to maximum result length of XMLTABLE function

In Version 10, the maximum length of a VARCHAR data type result column of the XMLTABLE function is changed from 32767 to the maximum length of a VARCHAR.

Change to how a positive, signed integer in an ORDER BY clause is treated

Explanation

Beginning in Version 10 conversion mode (from both Version 8 and Version 9.1), a positive, signed integer in an ORDER BY clause is treated as a *sort-key-expression*. Such integers were previously interpreted as column numbers.

For example, in previous versions of DB2, ORDER BY +1 in the following SELECT statement meant order by column 1 (C1).

```
SELECT C1, C2 FROM T1 ORDER BY +1;
```

Starting in Version 10, +1 means the constant +1, which has no effect on the order of the rows.

Possible impact to your DB2 environment

This change might cause unexpected results of queries that contain an ORDER BY clause with a positive, signed integer. However, no error is issued when such queries are run.

Actions to take

To prepare for this change, identify any queries that use a positive, signed integer in an ORDER BY clause to refer to a column in the result table. Modify these queries to use unsigned integers to identify column numbers.

Related reference:

 [order-by-clause \(DB2 SQL\)](#)

Binding DBRMs directly into plans is no longer supported

For pre-existing plans that are bound from DBRMs, you can use the COLLID parameter of the REBIND PLAN command to create packages. If you execute a plan that is bound from DBRMs, DB2 performs an automatic rebind that creates packages from the DBRMs and binds those packages into a plan. However, the recommendation is to use REBIND with the COLLID option so that you can specify bind options and receive more diagnostic information. If the installation uses the RACF access control module, owners of plans with DBRMs need to explicitly rebind the plans to convert the DBRMs to packages.

Some BIND PLAN and REBIND PLAN command options are no longer supported

The ACQUIRE(ALLOCATE) option of the BIND PLAN and REBIND PLAN commands is no longer supported. If you specify ACQUIRE(ALLOCATE), DB2 issues a warning message and uses ACQUIRE(USE).

Also, the MEMBER option of BIND PLAN and REBIND PLAN is no longer supported. If you specify MEMBER, DB2 issues a warning message, binds the specified DBRM into a package, and binds the package into a plan.

Plans and packages should be converted to DRDA protocol

Plans and packages that were previously bound using DBPROTOCOL(PRIVATE) should be converted to DRDA protocol before migration to Version 10. In Version

10, plans and packages that were bound with the DBPROTOCOL(PRIVATE) bind option and access remote locations cannot run. Applications that use packages or plans that were bound with DBPROTOCOL(PRIVATE) and access remote locations fail with SQLCODE -904. A rebind of those plans and packages must be explicitly performed before they can execute successfully. Job DSNTIJPM identifies the objects that must be converted to use DRDA protocol.

Change to GRANT statement

The PUBLIC AT ALL LOCATIONS clause is no longer allowed in the GRANT statement for table and view privileges as an alternative to PUBLIC. The DSNTPPCK program in Version 8 and Version 9.1 analyzes the embedded SQL statements in packages and plans for private protocol SQL, which is invalid in Version 10. The program produces a report that states which packages and member DBRMs of plans contain the invalid syntax. The program scans either the SYSIBM.SYSSTMT catalog table, the SYSIBM.SYSPACKSTMT catalog table, or both. Applications that issue dynamic SQL statements with the invalid PUBLIC AT ALL LOCATIONS clause will receive SQLCODE -199.

Change to IMMEDIATE option of BIND PACKAGE command

In DB2 Version 10, if IMMEDIATE is not specified on a BIND PACKAGE command, the default is "I", or INHERITFROMPLAN. In previous versions, the default was NO.

Changes to conversion of special characters in collection IDs and package names

In Version 10, DRDA character type parameter data is sent between client drivers and DB2 in UTF-8 Unicode, if those client drivers also have this support. Character type parameter data includes package names and collection IDs. Prior to Version 10, collection IDs and package names were sent in EBCDIC, and then converted to Unicode before being stored in the DB2 catalog. A collection ID or package name that is in the DB2 catalog from a bind that was initiated by an older driver might not match an ID or name that is sent to DB2 by a newer driver. This mismatch, which is caused by the way that some special characters are converted, can cause package-not-found errors. Job step DSNTGEN of job DSNTIJPM identifies package names and collection IDs that contain special characters that cause mismatches. From the upgraded client drivers, those packages need to be bound with the ACTION(REPLACE) option.

Changes to the RELEASE bind option

In releases prior to Version 10, the RELEASE bind option had no effect on database access threads. Starting in Version 10, by default, DB2 honors the RELEASE bind option for database access threads. You can modify this behavior by using the new MODIFY DDF PKGREL command.

Database metadata stored procedures are converted to Unicode

In DB2 Version 10, DB2-supplied database metadata stored procedures are encoded in Unicode. The Version 10 migration process redefines the stored procedures to use the following new load modules:

Table 3. Metadata stored procedures and associated load modules

Stored procedure	Load module
SYSIBM.SQLCOLPRIVILEGES	DSNACPRU
SYSIBM.SQLCOLUMNS	DSNACOLU
SYSIBM.SQLFOREIGNKEYS	DSNAFNKU
SYSIBM.SQLFUNCTIONCOLS	DSNAFCOU
SYSIBM.SQLFUNCTIONS	DSNAFUNU
SYSIBM.SQLGETTYPEINFO	DSNATYPU
SYSIBM.SQLPRIMARYKEYS	DSNAPRKU
SYSIBM.SQLPROCEDURECOLS	DSNAPCOU
SYSIBM.SQLPROCEDURES	DSNAPRCU
SYSIBM.SQLSPECIALCOLUMNS	DSNASPCU
SYSIBM.SQLSTATISTICS	DSNASTAU
SYSIBM.SQLTABLEPRIVILEGES	DSNATBPU
SYSIBM.SQLTABLES	DSNATBLU
SYSIBM.SQLUUDTS	DSNAUDTU

Some applications call the database metadata stored procedures to retrieve double-byte (DBCS) data. Those applications must be modified to use Unicode if they are bound on a DB2 server that has an EBCDIC SBCS CCSID, and the MIXED parameter is set to NO in the application defaults load module, *dsnhdecpl*.

AUTHID is the default owner of packages that are bound by DSNTRIN

In previous releases, the stored procedures and user-defined functions that are provided as part of the DB2 base product (DB2-supplied routines) were created and bound by processing inline DDL and bind statements in DSNTIJSJ and other installation jobs. By default, the ID that was used to run the job was also the authorization ID for creating the routines and the default package owner for those routines.

In Version 10, DB2-supplied routines are created and bound by running program DSNTRIN in job DSNTIJRT. The AUTHID parameter of DSNTRIN specifies the authorization ID for creating the routines. This ID is also the default owner of packages that are bound for those routines.

New default DEFINE attribute for dependent objects

As of Version 10, if the DEFINE attribute is not specified in the CREATE statement for explicitly created dependent objects (auxiliary indexes, XML indexes, and base table indexes), DB2 uses the DEFINE attribute of the base table space.

Exception: If the DEFINE attribute is not specified for explicitly created LOB table spaces, there is no correlation with the base table space until the auxiliary table is created. The DEFINE attribute is not inherited from the base, and the default is DEFINE YES.

Implicitly created dependent objects (base table indexes, LOB and XML table spaces, and their dependent indexes) inherit the DEFINE attribute of the base table

space if it is DEFINE NO. Otherwise, if the base table space attribute is DEFINE YES, the value of the IMPDSDEF subsystem parameter is used for the dependent objects.

Change for creating partitioned table spaces

To create a partitioned (non-universal) table space in DB2 Version 10 new-function mode, you must specify SEGSIZE 0 and the NUMPARTS keyword of the CREATE TABLESPACE statement. Before new-function mode, do not specify the SEGSIZE to create a partitioned table space.

Change to default for CREATE TABLESPACE statements

In previous releases of DB2, if a CREATE TABLESPACE statement contains the NUMPARTS clause but neither the MAXPARTITIONS clause nor the SEGSIZE clause, a partitioned (non-universal) table space is created. Beginning in Version 10 new-function mode, the same statement results in a range-partitioned table space with a segment size of 32 by default. You might observe this difference when a subsequent CREATE INDEX statement with the specified ranges fails because index partitioning of a range-partitioned table space is not supported.

In response, you can change the default segment size through the DSN6SYSP.DPSEGSZ subsystem parameter, which is externalized as the DEFAULT PARTITION SEGSIZE field on panel DSNTIP7. The value of the DPSEGSZ parameter can range from 0 to 64 in increments of 4; for example, 0, 4, 8, and so on up to 64. The default is 32.

When DPSEGSZ is set to 0, a CREATE TABLESPACE statement that contains the NUMPARTS clause but neither the MAXPARTITIONS clause nor the SEGSIZE clause results in a partitioned (non-universal) table space, which is the behavior of previous releases. Note that the DPSEGSZ parameter is provided for compatibility only. It is deprecated in Version 10, and you should take steps to modify affected CREATE TABLESPACE statements or plan to use partition-by-growth table spaces instead of partitioned (non-universal) table spaces.

Change to default SEGSIZE value for universal table spaces

In DB2 Version 10, the default SEGSIZE value for universal table spaces has changed from 4 to 32.

Upgrade to supported COBOL and PL/I compilers

If you use the Version 10 precompiler, you must upgrade to COBOL compilers that DB2 10 for z/OS supports. See the DB2 Program Directory for information about supported compilers. The generated SQLCA and SQLDA for COBOL have changed. The generated attribute for binary data items is now COMP-5 instead of COMP-4 or COMP. Applications that are compiled on compilers that do not support the COMP-5 attribute no longer work.

For some COBOL and PL/I compilers that are no longer supported, you can use a version of the precompiler that allows you to precompile applications that have dependencies on these unsupported compilers. You can use this version of the precompiler with the following unsupported compilers:

- OS/VS COBOL V1.2.4
- OS PL/I 1.5 (PL/I Opt. V1.5.1)
- VS/COBOL II V1R4
- OS PL/I 2.3

The load module for this precompiler is DSNHPC7. This precompiler is meant only to ease the transition from unsupported compilers to supported compilers. This precompiler has the following restrictions:

- There is no corresponding DB2 coprocessor function to match this precompiler.
- The precompiler does not support SQL procedures.
- Only COBOL and PL/I are supported.
- The SQL flagger is not supported.
- The precompiler produces Version 7 DBRMs, and does not support any capability that is newer than Version 7.
- The application defaults module must be named DSNHDECP.

Support of this precompiler is deprecated in Version 10.

GRAPHIC and NOGRAPHIC SQL processing options are removed

If you specify the SQL processing options GRAPHIC or NOGRAPHIC, DB2 issues a standard warning message about an invalid option. These options are superseded by the CCSID SQL processing option.

SELECT FROM data change statements in BEFORE triggers no longer supported

The following statements are no longer allowed in the body of a BEFORE trigger:

- SELECT FROM DELETE
- SELECT FROM INSERT
- SELECT FROM MERGE
- SELECT FROM UPDATE

RETURN statement in scalar functions must follow *option-list*

In previous DB2 versions, RETURN statements in scalar functions could be in any order, relative to other clauses. After migration to Version 10 conversion mode (from Version 8 or Version 9.1), the RETURN statement in CREATE FUNCTION or ALTER FUNCTION statements must follow *option-list*. If a RETURN statement precedes *option-list* in one of these statements, DB2 issues SQLCODE -199.

Changed behavior of LOCATE_IN_STRING function

In Version 10, a negative value for *start* in the LOCATE_IN_STRING function results in the search starting at the end of the source string. When *start* is negative, the starting position is $\text{LENGTH}(\text{source-string}) + \text{start} + 1$.

Changes to ROUND_TIMESTAMP and TRUNC_TIMESTAMP functions

In Version 10, the ROUND_TIMESTAMP and TRUNC_TIMESTAMP functions return the first day of the first ISO week of the ISO year when an ISO year format is specified.

Also, whenever a CC or SCC format is specified, a start of a century is considered to be year 01.

Changes to result of NEXT_DAY function

In Version 10, the result data type for the NEXT_DAY function is determined from the input data. If the first input argument is a string, the result is TIMESTAMP(6) WITHOUT TIME ZONE. Otherwise, the data type of the result is the same as the data type of the first input argument. For example, if the input is a date data type, the result is also a date data type. The following rules apply to the change in the NEXT_DAY function:

- Existing view definitions that reference the NEXT_DAY function should be regenerated with an ALTER VIEW statement.
- Existing materialized query statements that reference the NEXT_DAY function should be dropped and re-created.
- Bound SQL statements that reference the NEXT_DAY function only use the modified rules to determine the result data type if the statements are bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- DESCRIBE statements only use the modified rules to determine the result data type for the NEXT_DAY function if the statements are bound in Version 10 conversion mode (from Version 8 or Version 9.1) or later.
- Existing indexes that involve an expression that reference the NEXT_DAY function should be dropped and re-created.

Changes to MONTHS_BETWEEN function

In previous releases, the MONTHS_BETWEEN function ignored the time portion of arguments. In Version 10, the MONTHS_BETWEEN function takes the time portion of arguments into consideration.

Changes to TIMESTAMPDIFF function

In previous releases, the TIMESTAMPDIFF function allowed string input values that had more than six digits to the right of the decimal point. In Version 10, an error is issued if the string input value for TIMESTAMPDIFF function has more than six digits to the right of the decimal point.

Static SQL applications that use parallelism

DB2 incrementally rebinds the statements that use parallelism after migration to Version 10. Incremental rebinds can cause performance degradation. If you use the access control authorization exit (DSNX@XAC) for authorization, incremental rebinds can cause authorization failures because they trigger authorization checks of static SQL statements when the package is executed. The authorization checks are performed on the primary authorization ID during incremental rebinds.

You should manually rebind those statements that use parallelism after migration. You can run a query in job DSNTIIPM before you migrate to determine which statements can use parallelism, and are therefore candidates for incremental rebinds. You should consider rebinding those statements after migration, as soon as your Version 10 system is stable. After you migrate to Version 10, you can also run a performance trace, class 3 or class 10 for IFCID 360, to identify the plans and packages that contain static SQL queries that use parallelism, and therefore need to be rebound.

Enforced SELECT authorization checking for UPDATE and DELETE statements

DB2 Version 10 checks for the SELECT privilege or appropriate administrative privilege before allowing a user to execute UPDATE or DELETE statements that

reference an existing value in the target table. This authorization checking applies regardless of how the statements are executed (for example, bound in a package or executed in a dynamic statement). The authorization now also applies regardless of the setting of the SQLRULES(STD) bind option for static statements or the CURRENT RULES special register for dynamic statements. If the user does not have the necessary SELECT authorization or administrative privilege, a negative SQLCODE is returned.

Increased limit for work file record length

In Version 10 new-function mode, the limit for the row length in the result of a JOIN or the row length of a SORT record is increased from 32 767 bytes (1 page) to 65 529 bytes. The sort key maximum length is also increased from 16 000 bytes to 32 000 bytes. Applications that exceed the old limits fail in conversion mode (from both Version 8 and Version 9.1) with SQLCODE -670 or SQLCODE -136.

New restrictions for EXPLAIN tables

In DB2 Version 10 conversion mode (from both Version 8 and Version 9.1), EXPLAIN tables must be in Version 8 or later format and preferably encoded in Unicode. When EXPLAIN tables are in a format prior to the Version 8 format, DB2 returns SQLCODE -20008 reason code 2 for statements or commands that invoke EXPLAIN processing. Statements or commands that invoke EXPLAIN processing return SQLCODE +20520 reason code 2 if an EXPLAIN table is in Version 8 or Version 9.1 format, regardless of the encoding type. If an EXPLAIN table is in Version 10 format and encoded in EBCDIC, SQLCODE -878 is returned. When you convert EXPLAIN tables to Unicode encoding, applications that join with EXPLAIN tables might have different results because of the CCSID conversion. For more information, see Objects with different CCSIDs in the same SQL statement (DB2 Internationalization Guide).

Recommendation: Before you begin migration to Version 10, convert all EXPLAIN tables to the current version (8 or 9.1) format and Unicode encoding, and then check for joins to those tables. You can use job DSNTIJXA to convert most tables to current release format. You can use jobs DSNTIJXB and DSNTIJXC to migrate EBCDIC-encoded EXPLAIN tables to Unicode.

MEMBER CLUSTER table spaces indicated by MEMBER_CLUSTER column

In previous versions of DB2, a “K” or “I” in the TYPE column of the SYSTABLESPACE catalog table indicated that the table space had MEMBER CLUSTER structure. In Version 10, a new MEMBER_CLUSTER column on the SYSTABLESPACE catalog table is populated during the enabling-new-function mode (from both Version 8 and Version 9.1) migration process. For existing MEMBER CLUSTER table spaces, values of “K” in the TYPE column of SYSTABLESPACE are replaced with “L”, and values of “I” are replaced with blank. The MEMBER_CLUSTER column is populated with “Y”. After migration to enabling-new-function mode (from Version 8 or Version 9.1), applications that query “K” or “I” in the TYPE column must query the new MEMBER_CLUSTER column instead.

Changed values for the modification level in the product signature

The DB2 Version 10 product signature has the form DSN1001*m*, where *m* is the modification level. Values 0 and 1 are reserved for maintenance levels in

conversion mode from Version 8, conversion mode* from Version 8, enabling-new-function mode from Version 8, and enabling-new-function mode* from Version 8. Values 2 and 3 are for maintenance levels in conversion mode from Version 9.1, conversion mode* from Version 9.1, enabling-new-function mode from Version 9.1, and enabling-new-function mode* from Version 9.1. Values 5, 6, 7, 8, and 9 are for maintenance levels in new-function mode. Value 4 is undefined.

Changed behavior for the CREATE FUNCTION statement

In all forms of the CREATE FUNCTION statement, a parameter list is required. Functions without parameters must have empty parentheses specified, as in the following example: CREATE FUNCTION F1 () RETURNS INT RETURN 1.

In previous releases, if you specified CREATE FUNCTION without a parameter list (for example, CREATE FUNCTION F1 RETURNS INT RETURN 1), you received no errors. In Version 10, if you specify CREATE FUNCTION without a parameter list, DB2 issues an error.

Also, a CREATE FUNCTION statement for a non-inline SQL scalar function cannot have a parameter that is a distinct type that is based on a LOB data type. The inline SQL scalar functions have supported, and will continue to support, distinct type parameters where the underlying base data type is a LOB. For the syntax of this statement, see CREATE FUNCTION in the DB2 for z/OS SQL Reference Guide.

Different SQLSTATE returned for some DELETE or UPDATE statements

In previous releases of DB2, a SQLSTATE value of "00000" was returned for some DELETE or UPDATE statements that did not contain a WHERE clause, when SQLWARN flags were set. Those situations occurred during bind processing. As of DB2 Version 10, a SQLSTATE value of "01504" is returned in those situations. This change does not affect SQL statements that are executed on behalf of a DB2 for z/OS requester.

Changed default behavior of multiple-row inserts for ODBC z/OS applications

As of DB2 Version 10, the default behavior for multiple-row inserts is changed from non-atomic to atomic. To change the default behavior back to non-atomic, set keyword PARAMOPTATOMIC = 0 in the data source section of the ODBC initialization file.

Changes to ALTER TABLESPACE statement error codes

Before Version 10 conversion mode (from Version 8 or Version 9.1), if you execute an ALTER TABLESPACE MAXPARTITIONS statement on catalog objects, DB2 issues SQLCODE -644. After you migrate to Version 10 conversion mode (from Version 8 or Version 9.1), DB2 issues SQLCODE -607.

Also before Version 10 conversion mode (from Version 8 or Version 9.1), if you execute an ALTER TABLESPACE MAXPARTITIONS statement with a value of MAXPARTITIONS that is invalid with the page size and DSSIZE values, DB2 issues SQLCODE -4701. After you migrate to Version 10 conversion mode (from Version 8 or Version 9.1), DB2 issues SQLCODE -644.

Change to CREATE and ALTER statements

CREATE and ALTER statements for native SQL procedures no longer support the DEFAULT keyword for the SQL PATH option.

Change to ALTER PROCEDURE statement

When the REGENERATE option is specified for the ALTER PROCEDURE statement for native SQL procedure, any existing comment in the catalog for the routine is retained. In previous DB2 versions, the comment was cleared in this situation.

Change to DESCRIBE statement

In Version 10, when a DESCRIBE statement is used on a result table that includes a distinct type, it will now return information about the distinct type.

New restrictions on using DSNTIAUL

DSNTIAUL can no longer be used to process CREATE FUNCTION (SQL scalar) statements that would result in a package or CREATE TRIGGER statements. DSNTIAUL also cannot be used to process any other statement that contains *SQL-routine-body*. These statements are CREATE PROCEDURE (SQL external), CREATE PROCEDURE (SQL native), CREATE FUNCTION (SQL table), ALTER PROCEDURE (SQL native) with an ADD or REPLACE clause, and ALTER FUNCTION (SQL scalar) with an ADD or REPLACE clause.

Changes to SYSROUTINES

As of Version 10, the CREATEDTS column of SYSIBM.SYSROUTINES will always reflect the time that a CREATE statement was first issued for a routine. In previous releases of DB2, the CREATEDTS column might have contained different values if multiple versions of a routine were created.

Catalog restructured

In Version 10, the DB2 catalog is restructured to reduce lock contention. As a result, binding takes longer in Version 10 than it did in previous versions. Also, execution of the DECLARE GLOBAL TEMPORARY TABLE statement takes longer in Version 10.

Changed data type for an untyped parameter marker

In previous releases, an untyped parameter marker for a unary minus operator has an assumed DOUBLE data type. In DB2 Version 10, the assumed data type is DECFLOAT(34).

Changes to handling of special values Infinity, sNaN, and NaN

In previous releases, when DB2 returns a decimal floating-point (DECFLOAT) value for Infinity, NaN, or sNaN as a CHAR or VARCHAR string, the string is in mixed case. In DB2 Version 10, the values are returned in upper case as follows: INFINITY, NAN, or SNAN.

Changes for INSTEAD OF triggers

Changes are introduced for existing INSTEAD OF triggers that were defined on a view that has a ROWID column or a column that is based on an underlying column of any of the following types:

- A security label column.

- A row change timestamp column.
- A column that is defined with AS ROW BEGIN.
- A column that is defined with AS ROW END.
- A column that is defined with AS TRANSACTION START ID.

For such INSTEAD OF triggers, the following actions now fail with a negative SQLCODE:

- Rebinding the INSTEAD OF trigger package.
- Dropping and re-creating the INSTEAD OF trigger.
- Running the REPAIR DBD utility on a database that includes the INSTEAD OF trigger.
- Issuing ALTER TABLE ALTER COLUMN SET DATA type on a table that has a view on which the INSTEAD OF trigger is defined.

Change to positioned update or delete statements

Explanation

In DB2 Version 10, packages must be rebound if both of the following statements are true:

- The package contains static cursors that were bound in a previous version.
- The package contains dynamic UPDATE or DELETE statements that contain the WHERE CURRENT of clause. The WHERE CURRENT of clause indicates a positioned UPDATE or DELETE statement.

Possible impact to your DB2 environment

If any such packages are not rebound in Version 10, positioned UPDATE and DELETE statements fail with SQLCODE -20249 when they run against a down-level cursor.

Actions to take

After migration to Version 10 conversion mode (from Version 8 or Version 9.1), rebind packages that contain static cursors with positioned UPDATE or DELETE statements.

Change to stored procedure parameter values returned to non-Java clients

In previous releases, when a remote application calls a DB2 for z/OS stored procedure, the data types of the returned output data match the data types of the corresponding CALL statement arguments. Starting in Version 10 conversion mode (from both Version 8 and Version 9.1), the data types of the returned output data match the data types of the parameters in the stored procedure definition. This change can cause conversion failures for some applications that use non-Java client drivers such as .NET to call stored procedures on DB2. To prepare for this change, modify the CALL requests of your non-Java client applications to specify argument types that conform to the semantics of the client driver. If you want to temporarily override the Version 10 behavior, you can set the DDF_COMPATIBILITY subsystem parameter to SP_PARAMS_NJV.

The SP_PARMS_NJV option of the DDF_COMPATIBILITY subsystem parameter is deprecated. Although the option is supported in Version 10, it will be removed in a later release of DB2.

Change to results of JDBC method PreparedStatement.setTimestamp

TIMESTAMP WITH TIME ZONE is first supported in DB2 for z/OS Version 10 new-function mode. Before Version 10 new-function mode, if the value that is assigned to a column (the second parameter of PreparedStatement.setTimestamp) has the java.sql.Timestamp data type, and the column data type is not known, the IBM Data Server Driver for JDBC and SQLJ chooses TIMESTAMP as the target data type. However, starting with DB2 for z/OS Version 10 new-function mode, unless the value that is assigned to the column is 0001-01-01-00:00:00.000000 or 9999-12-31-23:59:59.999999, the driver chooses TIMESTAMP WITH TIME ZONE as the column data type. If the driver chooses the TIMESTAMP data type, and the column type is actually TIMESTAMP WITH TIME ZONE, the database manager sets the time zone in the target column using the value of the IMPLICIT_TIMEZONE DECP value. This value might differ from the value that is inserted prior to Version 10 new-function mode.

To produce the same results before and after new-function mode when PreparedStatement.setTimestamp is executed, specify a com.ibm.db2.jcc.DBTimestamp value as the second parameter.

Change to behavior of comma operator in XQuery path expression

A comma operator in a DB2 XQuery path expression results in SQLCODE -16031. In previous releases, a comma operator in the DB2 XQuery path expression predicate was interpreted as the and operator.

Change in how DB2 returns stored procedure output parameter data to remote clients

When an application on a client system calls a stored procedure on a DB2 10 for z/OS server, DB2 now handles the output parameters differently. Previously, DB2 returned stored procedure output parameters that were formatted according to the SQL type of the corresponding argument in the CALL statement. DB2 10 now returns output parameters that are formatted according to the SQL type of the corresponding parameter in the stored procedure declaration. This new behavior provides improved performance at the server by avoiding unnecessary server data conversions. Also, this new behavior is consistent with the existing server behavior for the return of query and select output data and it is consistent with the behavior of other DB2 family servers. Therefore, it provides applications with a more consistent and predictable interface to DB2.

In general, for applications that conform to client standards, this change has no impact to the calling application. In some cases, however, application changes might be needed. To prepare for this change, examine your remote applications that call DB2 stored procedures. If necessary, modify the CALL statements in your remote applications to specify argument data types that match the data types of the parameters in the stored procedure definitions.

When the change in behavior occurs

In general, the new behavior occurs after migration to DB2 10 for z/OS conversion mode from Version 8 (CM8) or conversion mode from Version 9.1 (CM9). However, the following exceptions apply:

- For applications using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to access a DB2 10 data sharing group in CM8 or CM9, where the `enableSysplexWLB` client property is set to true:

In such a configuration, the change in behavior occurs when the data sharing group is migrated to DB2 10 new-function mode (NFM). As long as the data sharing group is in CM8 or CM9 to support coexistence, DB2 uses the old behavior. The old behavior is used because the IBM Data Server Driver for JDBC and SQLJ might be caching data descriptors for outputs from the server. When the `enableSysplexWLB` property is set to true, the old behavior is maintained to ensure that the cached client descriptors are valid regardless of which member of the data sharing group is accessed.

To prepare for this change in behavior while DB2 10 is in CM8 or CM9 to support coexistence, test your applications with the `enableSysplexWLB` client property set to false. Or, test the applications against a stand-alone DB2 10 server, if such a system is available for testing. The new DB2 10 behavior will be used for each of those applications even though the DB2 data sharing group is in CM8 or CM9.

- For applications using the IBM Data Server Driver for JDBC and SQLJ or non-Java clients (such as .NET), Version 9 or earlier:

In such a configuration, the change in behavior occurs, by default, after migration to DB2 10 for z/OS CM8 or CM9. However, the change in behavior can be temporarily deferred. To temporarily override the DB2 10 behavior for returning stored procedure output parameter data to the IBM Data Server Driver for JDBC and SQLJ or non-Java clients only (such as .NET), set the `DDF_COMPATIBILITY` subsystem parameter to `SP_PARMS_NJV`. The `SP_PARMS_NJV` option of the `DDF_COMPATIBILITY` subsystem parameter is deprecated. Although the option is supported in DB2 10, it will be removed in a later release of DB2. For more information about `DDF_COMPATIBILITY`, see Subsystem parameters that are not on installation panels (DB2 Installation and Migration).

Examples of application impacts

The following examples use the IBM Data Server Driver for JDBC and SQLJ and .NET. Other drivers might behave differently. Because DB2 10 no longer converts data types when returning stored procedure output parameter data, there might be similar impacts to applications for other data types that are not discussed in the examples. If the calling application specifies arguments that are different from the declared parameter types, then the mismatch between the argument and parameter data types is handled by the client system, according to the client's programming model and the client's capabilities. If the client performs data type conversions such as for numeric data, the result of the conversion performed by the client might differ from the result that was previously returned when the DB2 server performed the conversion.

Example 1

A stored procedure parameter is declared as an `INTEGER` data type, but the application `CALL` statement specifies a `SMALLINT` argument.

- Behavior before DB2 10: DB2 converts the `INTEGER` data and returns a 2-byte `SMALLINT` value, which maps to an `Int16` .NET data type.

- New behavior in DB2 10: DB2 returns a 4-byte INTEGER value, which maps to an Int32 .NET data type.

Sample application impact (.NET): If the application uses an IBM Data Server .NET client, the stored procedure CALL statement now fails with an invalid conversion error because the Int16 and Int32 .NET data types are not compatible with each other. The application must be changed to specify an Int32 argument in the CALL statement as required by the .NET programming model. Applications that are coded with an Int32 argument and call stored procedures that have an INTEGER parameter require no change.

The .NET programming model enforces strong data typing. Thus, even though the argument type and the parameter type are compatible SQL types, the .NET driver enforces strong data type checking according to the .NET semantics. Refer to the .NET documentation for information about .NET strong type checking.

Example 2

A stored procedure parameter is declared as a REAL data type, but the application CALL statement specifies a DOUBLE argument.

- Behavior before DB2 10: DB2 converts the output data to a DOUBLE value before returning it to the client.
- New behavior in DB2 10: DB2 returns the data to the client as a REAL value.

Sample application impact (Java): If an application that uses the IBM Data Server Driver for JDBC and SQLJ uses the registerOutParameter() method to register the output parameter as a DOUBLE value before calling a stored procedure that has a REAL parameter, the following behavior occurs if the Java application uses the getObject() method to retrieve the output value:

- Before Version DB2 10, DB2 converted the REAL value to a DOUBLE value and returned it to the client. The IBM Data Server Driver for JDBC and SQLJ returned the DOUBLE value directly to the application as the result of the getObject() method.
- Beginning in DB2 10, DB2 returns the REAL value to the client. The IBM Data Server Driver for JDBC and SQLJ converts the REAL value to a DOUBLE value and returns it to application as the result of the getObject() method. APAR IC80974 for LUW clients and APAR PM58951 (JCC 3.63.131) and PM58952 (JCC 4.13.136) for z/OS clients are required in the IBM Data Server Driver for JDBC and SQLJ to ensure that the conversion is completed by the getObject() method. If the APAR fix is not applied, the Java application gets a ClassCastException.

Example 3

A stored procedure parameter is declared as a TIMESTAMP data type, but the application CALL statement specifies a VARCHAR argument.

- Behavior before DB2 10: DB2 converts the fixed-length TIMESTAMP data and returns a 26-byte VARCHAR value, containing a TIMESTAMP value in ISO format (*yyyy-mm-dd-hh.mm.ss[.ffffff]*). The VARCHAR value maps to a String .NET data type.
- New behavior in DB2 10: DB2 returns a 26-byte fixed-length TIMESTAMP value, containing a TIMESTAMP value in ISO format. The TIMESTAMP value maps to a DateTime .NET data type.

Sample application impact (Java): If the application uses the IBM Data Server Driver for JDBC and SQLJ, then an application change might be required, depending on what method the Java application uses to retrieve the parameter data.

- Before DB2 10, the `getString()` method previously returned `TIMESTAMP` data in ISO format, while the `getTimeStamp()` method returned `TIMESTAMP` data in Java format
- Beginning in DB2 10, both the `getString()` and `getTimeStamp()` methods return the `TIMESTAMP` value in Java format (`yyyy-mm-dd hh:mm:ss[.fffffffff]`).

An application change might be required if the Java application uses the `getString()` method and the application depends on receiving `TIMESTAMP` values in ISO format. Applications that use the `getTimeStamp()` method require no change.

Similar considerations apply for `TIME` parameters. Beginning in DB2 10, DB2 returns `TIME` parameters as `TIME` data values in ISO format (`hh.mm.ss`), regardless of the SQL type of the corresponding argument in the `CALL` statement. For Java applications, the `getString()` method now returns `TIME` data in Java format (`hh.mm.ss`). If a Java application uses the `getString()` method and requires `TIME` data in ISO format, the application must be examined for possible changes.

Change to IBM Data Server Driver for JDBC and SQLJ handling of `TIMESTAMP WITH TIME ZONE` data type

Before DB2 Version 10 new-function mode, the `TIMESTAMP WITH TIME ZONE` data type was not supported. If a Java client application passed a timestamp input value to a `TIMESTAMP` column, the IBM Data Server Driver for JDBC and SQLJ did not include the local time zone with the timestamp value. Starting with DB2 Version 10 new-function mode, the `TIMESTAMP WITH TIME ZONE` data type is supported. If a Java client application passes a timestamp input value to a `TIMESTAMP` column, the IBM Data Server Driver for JDBC and SQLJ constructs a timestamp input value that includes the local time zone. If the value that the driver sends to the server is out of supported range for the server, the application receives `SQLCODE -181`. You can temporarily prevent this error by including `IGNORE_TZ` in the settings for the `DDF_COMPATIBILITY` subsystem parameter.

Changes to datetime built-in functions

Explanation

Many datetime functions allow arguments containing *string representations of datetime values*. Valid formats for those strings are described in the DB2 SQL Reference.

In DB2 Version 10 conversion mode (from Version 8 or Version 9.1), the following additional string formats are allowed in the specified limited contexts:

- A string value of seven characters representing a date is allowed as an argument to the `DATE` function only.
- A string value of 8, 13, or 14 characters representing a point in time is allowed as an argument to the `TIMESTAMP` function only.

Possible impact to your DB2 environment

After migration to Version 10, applications that provide a seven-character string argument to represent a date for built-in functions other than the `DATE` function will return an error. Applications that provide a string value of 8, 13, or 14 characters to represent a point in time as an argument for built-in functions other than the `TIMESTAMP` function also return an error.

Actions to take

To prepare for this change, use valid string formats that are described in String representations of datetime values (DB2 SQL) in arguments for all of your datetime functions.

SQLCODE change for subsequent CAF CONNECT attempts

Explanation

In previous releases, a call attachment facility (CAF) CONNECT request that is followed by another CONNECT request without an intervening disconnect results in a zero return code. In DB2 Version 10, if the second CONNECT request is for a different or unknown subsystem, group attachment, or subgroup attachment name, a -924 SQLCODE is returned. If the second CONNECT request is for the same subsystem, group attachment, or subgroup attachment name, a +361 SQLCODE is returned. A failed attempt to connect to the CAF does not change the current connection. Therefore, in both of these scenarios, the existing CAF connection persists.

Possible impact to your DB2 environment

Some of your applications might receive a -924 or +361 return code where a zero return code was previously returned.

Actions to take

Review your applications for subsequent CONNECT requests for CAF. Modify these applications to handle the new SQLCODEs or rewrite the applications to remove subsequent CONNECT requests.

Delimiters used for accessing tables on DB2 for Linux, UNIX, and Windows

Explanation

DB2 Version 10 resolves aliases prior to sending SQL statements to a remote site for applications that use system-directed access. During a remote package bind against the remote site, modified SQL statement text is bound on the remote system. The DRDA_RESOLVE_ALIAS subsystem parameter is provided in DB2 Version 8 and DB2 Version 9.1 to help verify applications that are affected by this change of behavior before you migrate to DB2 Version 10.

Possible impact to your DB2 environment

This change can impact applications that access a DB2 for Linux, UNIX, and Windows server if the SQL preprocessing option QUOTE or QUOTESQL is used. QUOTE or QUOTESQL specifies that a quotation mark (") is used as the string delimiter and an apostrophe (') is used for SQL identifiers in SQL statements. This option does not control how the COBOL compiler processes string delimiters within the application program statements. DB2 for Linux, UNIX, and Windows does not support statement strings that have been precompiled under the QUOTE or QUOTESQL option and returns a warning on the BIND command. DB2 for z/OS does use the precompiler option to govern which string delimiter to use for SQL identifiers when modifying the SQL text. This causes a bind or rebind to fail on DB2 for Linux, UNIX, and Windows in DB2 Version 10 when the QUOTE or

QUOTESQL precompiler option is used to generate the DBRM that is the source of the remote bind package processing.

Actions to take

When accessing a remote table on a DB2 for Linux, UNIX, and Windows server using an alias, applications must be precompiled using the APOST or APOSTSQL option. Character string literals must be delimited by apostrophes and SQL identifiers must be delimited by quotation marks.

Qualify user-defined function names

If you use a user-defined function that has the same name as a built-in function that has been added to Version 10, ensure that you fully qualify the function name. If the function name is unqualified and "SYSIBM" precedes the schema that you used for this function in the SQL path, DB2 invokes one of the built-in functions.

For a list of built-in functions, including those that have been added in Version 10, see Functions (DB2 SQL).

SQLCODE changes

Some SQLCODE numbers and message text might have changed in DB2 Version 10. Also, the conditions under which some SQLCODEs are issued might have changed.

SQL reserved words

Version 10 has several new SQL reserved words. Refer to Reserved words (DB2 SQL) for the list of reserved words, and adjust your applications accordingly.



Determining the value of any SQL processing options that affect the design of your program

When you process SQL statements in an application program, you can specify options that describe the basic characteristics of the program. You can also indicate how you want the output listings to look. Although most of these options do not affect how you design or code the program, a few options do.

SQL processing options specify program characteristics such as the following items:

- The host language in which the program is written
- The maximum precision of decimal numbers in the program
- How many lines are on a page of the precompiler listing

In many cases, you may want to accept the default value provided.

To determine the value of any SQL processing options that affect the design of your program:

Review the list of SQL processing options and decide the values for any options that affect the way that you write your program. For example, you need to know if you are using NOFOR or STDSQL(YES) before you begin coding.

Related concepts:

"DB2 program preparation overview" on page 973

Related reference:

“Descriptions of SQL processing options” on page 932

Changes that invalidate packages

Changes to your program or database objects can invalidate packages.

A change to your program probably invalidates one or more of your packages. For some changes, you must bind a new object; for others, rebinding is sufficient.

A package can also become invalid for reasons that do not depend on operations in your program. For example, when an index is dropped that is used in an access path by one of your queries, a package can become invalid. In those cases, DB2 might rebind the package automatically the next time that the package is used.

The following table lists the actions that you must take when changes are made to your program or database objects.

Table 4. Changes that require packages to be rebound.

Change made	Required action
Run RUNSTATS to update catalog statistics	Rebind the package by using the REBIND command. Rebinding might improve the access path that DB2 uses.
Add an index to a table	Rebind the package by using the REBIND command. Rebinding causes DB2 to consider using the index when accessing this table.
Change the bind options ¹	Rebind the package by using the REBIND command and specifying the new value for the bind option. If the option that you want to change is not available for the REBIND command, issue the BIND command with ACTION(REPLACE) instead.
Change both statements in the host language and SQL statements	Precompile, compile, and link the application program. Issue the BIND command with ACTION(REPLACE) for the package.
Drop a table, index, or other object, and re-create the object	If a table with a trigger is dropped, re-create the trigger if you re-create the table. Otherwise, no change is required. DB2 attempts to automatically rebind the package the next time it is run.
Drop an object that a package depends on	No action is required. If the package becomes invalid, DB2 automatically rebinds the package the next time that it is allocated.
Revoke an authorization to use an object	No action is required. DB2 attempts to automatically rebind the package the next time it is run. Automatic rebind fails if authorization is still not available. In this case, you must rebind the package by using the REBIND command.
Rename a column in a table on which a package is dependent	No action is required. DB2 automatically rebinds invalidated packages. If automatic rebind is unsuccessful, modify, recompile, and rebind the affected applications.

Table 4. Changes that require packages to be rebound. (continued)

Change made	Required action
RUN REPAIR DBD REBUILD on a database	Trigger packages in the database are invalidated. Rebind all trigger packages in the database
Convert a partitioned table space to a range-partitioned universal table space	No action is required. DB2 automatically rebinds invalidated packages. If automatic rebind is unsuccessful, modify, recompile, and rebind the affected applications.
Convert a simple table space to a partition-by-growth universal table space	No action is required. DB2 automatically rebinds invalidated packages. If automatic rebind is unsuccessful, modify, recompile, and rebind the affected applications.
ALTER TABLESPACE with BUFFERPOOL to change the buffer pool page size	No action is required. DB2 automatically rebinds invalidated packages. If automatic rebind is unsuccessful, modify, recompile, and rebind the affected applications.
ALTER TABLESPACE with MAXPARTITIONS to change the maximum number of partitions	No action is required. DB2 automatically rebinds invalidated packages. If automatic rebind is unsuccessful, modify, recompile, and rebind the affected applications.

Note:

1. In the case of changing the bind options, the change is not actually made until you perform the required action.

Related concepts:

“Automatic rebinding” on page 970


“Trigger packages” on page 504

Related tasks:


 [Checking for invalid packages \(DB2 Performance\)](#)

“Rebinding an application” on page 962

Related reference:

 [Invalid and inoperative packages \(Managing Security\)](#)

Related information:

 [00E30305 \(DB2 Codes\)](#)

Determining the value of any bind options that affect the design of your program

Several options of the BIND PACKAGE and BIND PLAN commands can affect your program design. For example, you can use a bind option to ensure that a package or plan can run only from a particular CICS connection or IMS region. Your code does not need to enforce this situation.

To determine the value of any bind options that affect the design of your program:

Review the list of bind options and decide the values for any options that affect the way that you write your program. For example, you should decide the values

of the ACQUIRE and RELEASE options before you write your program. These options determine when your application acquires and releases locks on the objects it uses.

Related reference:

 [BIND and REBIND options for packages and plans \(DB2 Commands\)](#)

Programming applications for performance

You can achieve better DB2 performance by considering performance as you program and deploy your applications.

To improve the performance of application programs that access data in DB2, use the following approaches when writing and preparing your programs:

- Program your applications for concurrency. The goal is to program and prepare applications in a way that:

- Protects the integrity of the data that is being read or updated from being changed by other applications.
- Minimizes the length of time that other access to the data is prevented.

For more information about DB2 concurrency and recommendations for improving concurrency in your application programs, see the following topics:

- Concurrency recommendations for application designers (Introduction to DB2 for z/OS)
 - Concurrency and locks (DB2 Performance)
 - Improving concurrency (DB2 Performance)
 - Improving concurrency in data sharing environments (DB2 Data Sharing Planning and Administration)
- Write SQL statements that access data efficiently. The predicates, subqueries, and other structures in SQL statements affect the access paths that DB2 uses to access the data.

For information about how to write SQL statements that access data efficiently, see the following topics:

- Ways to improve query performance (Introduction to DB2 for z/OS)
 - Writing efficient SQL queries (DB2 Performance)
- Use EXPLAIN or SQL optimization tools to analyze the access paths that DB2 chooses to process your SQL statements. By analyzing the access path that DB2 uses to access the data for an SQL statement, you can discover potential problems. You can use this information to modify your statement to perform better.

For information about how you can use EXPLAIN tables, and SQL optimization tools such as IBM Data Studio, to analyze the access paths for your SQL statements, see the following topics:

- Investigating access path problems (DB2 Performance)
- Using EXPLAIN to understand the access path (Introduction to DB2 for z/OS)
- Investigating SQL performance by using EXPLAIN (DB2 Performance)
- Interpreting data access by using EXPLAIN (DB2 Performance)
- EXPLAIN tables (DB2 Performance)
- EXPLAIN (DB2 SQL)
- Tuning SQL with Optim Query Tuner, Part 1: Understanding access paths (IBM developerWorks)

- Generating visual representations of access plans (IBM Data Studio)
- Consider performance in the design of applications that access distributed data. The goal is to reduce the amount of network traffic that is required to access the distributed data, and to manage the use of system resources such as distributed database access threads and connections.

For information about improving the performance of applications that access distributed data, see the following topics:

- Ways to reduce network traffic (Introduction to DB2 for z/OS)
- Managing DB2 threads (DB2 Performance)
- Improving performance for applications that access distributed data (DB2 Performance)
- Improving performance for SQL statements in distributed applications (DB2 Performance)
- Use stored procedures to improve performance, and consider performance when creating stored procedures.

For information about stored procedures and DB2 performance, see the following topics:

- Implementing DB2 stored procedures (DB2 Administration Guide)
- Improving the performance of stored procedures and user-defined functions (DB2 Performance)

Related concepts:

- ➡ Query and application performance analysis (Introduction to DB2 for z/OS)
- ➡ Programming for the instrumentation facility interface (IFI) (DB2 Performance)

Related tasks:

Chapter 1, “Planning for and designing DB2 applications,” on page 1

Chapter 3, “Coding SQL statements in application programs: General information,” on page 159

- ➡ Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Designing your application for recovery

If your application fails or DB2 terminates abnormally, you need to ensure the integrity of any data that was manipulated in your application. You should consider possible recovery situations when you design your application.

To design your application for recovery:

1. Put any changes that logically need to be made at the same time in the same unit of work. This action ensures that in case DB2 terminates abnormally or your application fails, the data is left in a consistent state.

A unit of work is a logically distinct procedure that contains steps that change the data. If all the steps complete successfully, you want the data changes to become permanent. But, if any of the steps fail, you want all modified data to return to the original value before the procedure began. For example, suppose two employees in the sample table DSN8A10.EMP exchange offices. You need to exchange their office phone numbers in the PHONENO column. You need to use two UPDATE statements to make each phone number current. Both statements, taken together, are a unit of work. You want both statements to

complete successfully. For example, if only one statement is successful, you want both phone numbers rolled back to their original values before attempting another update.

2. Consider how often you should commit any changes to the data.

If your program abends or the system fails, DB2 backs out all uncommitted data changes. Changed data returns to its original condition without interfering with other system activities.

For IMS and CICS applications, if the system fails, DB2 data does not always return to a consistent state immediately. DB2 does not process indoubt data (data that is neither uncommitted nor committed) until you restart IMS or the CICS attachment facility. To ensure that DB2 and IMS are synchronized, restart both DB2 and IMS. To ensure that DB2 and CICS are synchronized, restart both DB2 and the CICS attachment facility.

3. Consider whether your application should intercept abends.

If your application intercepts abends, DB2 commits work, because it is unaware that an abend has occurred. If you want DB2 to roll back work automatically when an abend occurs in your program, do not let the program or run time environment intercept the abend. If your program uses Language Environment[®], and you want DB2 to roll back work automatically when an abend occurs in the program, specify the run time options ABTERMENC(ABEND) and TRAP(ON).

4. **For TSO applications only:** Issue COMMIT statements before you connect to another DBMS.

If the system fails at this point, DB2 cannot know whether your transaction is complete. In this case, as in the case of a failure during a one-phase commit operation for a single subsystem, you must make your own provision for maintaining data integrity.

5. **For TSO applications only:** Determine if you want to provide an abend exit routine in your program.

If you provide this routine, it must use tracking indicators to determine if an abend occurs during DB2 processing. If an abend does occur when DB2 has control, you must allow task termination to complete. DB2 detects task termination and terminates the thread with the ABRT parameter. Do not re-run the program.

Allowing task termination to complete is the only action that you can take for abends that are caused by the CANCEL command or by DETACH. You cannot use additional SQL statements at this point. If you attempt to execute another SQL statement from the application program or its recovery routine, unexpected errors can occur.

Related concepts:

 Unit of work (Introduction to DB2 for z/OS)

Unit of work in TSO

Applications that use the TSO attachment facility can explicitly define units of work by using the SQL COMMIT and ROLLBACK statements.

In TSO applications, a unit of work starts when the first updates of a DB2 object occur. A unit of work ends when one of the following conditions occurs:

- The program issues a subsequent COMMIT statement. At this point in the processing, your program has determined that the data is consistent; all data changes that were made since the previous commit point were made correctly.

- The program issues a subsequent ROLLBACK statement. At this point in the processing, your program has determined that the data changes were not made correctly and, therefore, should not be permanent. A ROLLBACK statement causes any data changes that were made since the last commit point to be backed out.
- The program terminates and returns to the DSN command processor, which returns to the TSO Terminal Monitor Program (TMP).

The first and third conditions in the preceding list are called a commit point. A *commit point* occurs when you issue a COMMIT statement or your program terminates normally.

Related reference:

 COMMIT (DB2 SQL)

 ROLLBACK (DB2 SQL)

Unit of work in CICS

CICS applications can explicitly define units of work by using the CICS SYNCPOINT command. Alternatively, units of work are defined implicitly by several logic-breaking points.

All the processing that occurs in your program between two commit points is known as a logical unit of work (LUW) or unit of work. In CICS applications, a unit of work is marked as complete by a commit or synchronization (sync) point, which is defined in one of following ways:

- Implicitly at the end of a transaction, which is signaled by a CICS RETURN command at the highest logical level.
- Explicitly by CICS SYNCPOINT commands that the program issues at logically appropriate points in the transaction.
- Implicitly through a DL/I PSB termination (TERM) call or command.
- Implicitly when a batch DL/I program issues a DL/I checkpoint call. This call can occur when the batch DL/I program shares a database with CICS applications through the database sharing facility.

For example, consider a program that subtracts the quantity of items sold from an inventory file and then adds that quantity to a reorder file. When both transactions complete (and not before) and the data in the two files is consistent, the program can then issue a DL/I TERM call or a SYNCPOINT command. If one of the steps fails, you want the data to return to the value it had before the unit of work began. That is, you want it rolled back to a previous point of consistency. You can achieve this state by using the SYNCPOINT command with the ROLLBACK option.

By using a SYNCPOINT command with the ROLLBACK option, you can back out uncommitted data changes. For example, a program that updates a set of related rows sometimes encounters an error after updating several of them. The program can use the SYNCPOINT command with the ROLLBACK option to undo all of the updates without giving up control.

The SQL COMMIT and ROLLBACK statements are not valid in a CICS environment. You can coordinate DB2 with CICS functions that are used in programs, so that DB2 and non-DB2 data are consistent.

Planning for program recovery in IMS programs

To be prepared for recovery situations for IMS programs that access DB2 data, you need to make several design decisions that are specific to IMS programs. These decisions are in addition to the general recommendations that you should follow when designing your application for recovery.

Both IMS and DB2 handle recovery in an IMS application program that accesses DB2 data. IMS coordinates the process, and DB2 handles recovery for DB2 data.

To plan for program recovery in IMS programs:

1. For a program that processes messages as its input, decide whether to specify single-mode or multiple-mode transactions on the TRANSACT statement of the APPLCTN macro for the program.

Single-mode

Indicates that a commit point in DB2 occurs each time the program issues a call to retrieve a new message. Specifying single-mode can simplify recovery; if the program abends, you can restart the program from the most recent call for a new message. When IMS restarts the program, the program starts by processing the next message.

Multiple-mode

Indicates that a commit point occurs when the program issues a checkpoint call or when it terminates normally. Those two events are the only times during the program that IMS sends the program's output messages to their destinations. Because fewer commit points are processed in multiple-mode programs than in single-mode programs, multiple-mode programs could perform slightly better than single-mode programs. When a multiple-mode program abends, IMS can restart it only from a checkpoint call. Instead of having only the most recent message to reprocess, a program might have several messages to reprocess. The number of messages to process depends on when the program issued the last checkpoint call.

DB2 does some processing with single- and multiple-mode programs. When a multiple-mode program issues a call to retrieve a new message, DB2 performs an authorization check and closes all open cursors in the program.

2. Decide whether to issue checkpoint calls (CHKP) and if so, how often to issue them. Each call indicates to IMS that the program has reached a sync point and establishes a place in the program from which you can restart the program.

Consider the following factors when deciding when to use checkpoint calls:

- How long it takes to back out and recover that unit of work. The program must issue checkpoints frequently enough to make the program easy to back out and recover.
- How long database resources are locked in DB2 and IMS.
- For multiple-mode programs: How you want the output messages grouped. Checkpoint calls establish how a multiple-mode program groups its output messages. Programs must issue checkpoints frequently enough to avoid building up too many output messages.

Restriction: You cannot use SQL COMMIT and ROLLBACK statements in the DB2 DL/I batch support environment, because IMS coordinates the unit of work.

3. Issue CLOSE CURSOR statements before any checkpoint calls or GU calls to the message queue, not after.

4. After any checkpoint calls, set the value of any special registers that were reset if their values are needed after the checkpoint:

A CHKP call causes IMS to sign on to DB2 again, which resets the special registers that are shown in the following table.

Table 5. Special registers that are reset by a checkpoint call.

Special register	Value to which it is reset after a checkpoint call
CURRENT PACKAGESET	blanks
CURRENT SERVER	blanks
CURRENT SQLID	blanks
CURRENT DEGREE	1

5. After any commit points, reopen the cursors that you want and re-establish positioning
6. Decide whether to specify the WITH HOLD option for any cursors. This option determines whether the program retains the position of the cursor in the DB2 database after you issue IMS CHKP calls. You always lose the program database positioning in DL/I after an IMS CHKP call.

The program database positioning in DB2 is affected according to the following criteria:

- If you do not specify the WITH HOLD option for a cursor, you lose the position of that cursor.
 - If you specify the WITH HOLD option for a cursor and the application is message-driven, you lose the position of that cursor.
 - If you specify the WITH HOLD option for a cursor and the application is operating in DL/I batch or DL/I BMP, you retain the position of that cursor.
7. Use IMS rollback calls, ROLL and ROLB, to back out DB2 and DL/I changes to the last commit point. These options have the following differences:

ROLL

Specifies that all changes since the last commit point are to be backed out and the program is to be terminated. IMS terminates the program with user abend code U0778 and without a storage dump.

When you issue a ROLL call, the only option you supply is the call function, ROLL.

ROLLB

Specifies that all changes since the last commit point are to be backed out and control is to be returned to the program so that it can continue processing.

A ROLB call has the following options:

- The call function, ROLB
- The name of the I/O PCB

How ROLL and ROLB calls effect DL/I changes in a batch environment depends on the IMS system log and back out options that are specified, as shown in the following table.

Table 6. Effects of ROLL and ROLLB calls on DL/I changes in a batch environment

Rollback call	Options specified		Result
	System log option	Backout option	
ROLL	tape	any	DL/I does not back out updates, and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
	disk	BKO=NO	
	disk	BKO=YES	DL/I backs out updates, and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
ROLB	tape	any	DL/I does not back out updates, and an AL status code is returned in the PCB. DB2 backs out updates to the previous checkpoint. The DB2 DL/I support causes the application program to abend when ROLB fails.
	disk	BKO=NO	
	disk	BKO=YES	DL/I backs out database updates, and control is passed back to the application program. DB2 backs out updates to the previous checkpoint. Restriction: You cannot specify the address of an I/O area as one of the options on the call; if you do, your program receives an AD status code. However, you must have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB.

Related concepts:

“Checkpoints in IMS programs” on page 62

Unit of work in IMS online programs

IMS applications can explicitly define units of work by using a CHKP, SYNC, ROLL, or ROLB call, or, for single-mode transactions, a GU call.

In IMS, a unit of work starts when one of the following events occurs:

- When the program starts
- After a CHKP, SYNC, ROLL, or ROLB call has completed
- For single-mode transactions, when a GU call is issued to the I/O PCB

A unit of work ends when one of the following events occurs:

- The program issues either a subsequent CHKP or SYNC call, or, for single-mode transactions, a GU call to the I/O PCB. At this point in the processing, the data is consistent. All data changes that were made since the previous commit point are made correctly.
- The program issues a subsequent ROLB or ROLL call. At this point in the processing, your program has determined that the data changes are not correct and, therefore, that the data changes should not become permanent.
- The program terminates.

Restriction: The SQL COMMIT and ROLLBACK statements are not valid in an IMS environment.

A commit point occurs in a program as the result of any one of the following events:

- The program terminates normally. Normal program termination is always a commit point.
- The program issues a checkpoint call. *Checkpoint calls* are a program's means of explicitly indicating to IMS that it has reached a commit point in its processing.
- The program issues a SYNC call. A *SYNC call* is a Fast Path system service call to request commit-point processing. You can use a SYNC call only in a non-message-driven Fast Path program.
- For a program that processes messages as its input, a commit point can occur when the program retrieves a new message. This behavior depends on the mode that you specify in the APPLCTN macro for the program:
 - If you specify single-mode transactions, a commit point in DB2 occurs each time the program issues a call to retrieve a new message.
 - If you specify multiple-mode transactions or you do not specify a mode, a commit point occurs when the program issues a checkpoint call or when it terminates normally.

At the time of a commit point, the following actions occur:

- IMS and DB2 can release locks that the program has held since the last commit point. Releasing these locks makes the data available to other application programs and users.
- DB2 closes any open cursors that the program has been using.
- IMS and DB2 make the program's changes to the database permanent.
- If the program processes messages, IMS sends the output messages that the application program produces to their final destinations. Until the program reaches a commit point, IMS holds the program's output messages at a temporary destination.

If the program abends before reaching the commit point, the following actions occur:

- Both IMS and DB2 back out all the changes the program has made to the database since the last commit point.
- IMS deletes any output messages that the program has produced since the last commit point (for nonexpress PCBs).
- If the program processes messages, people at terminals and other application programs receive information from the terminating application program.

If the system fails, a unit of work resolves automatically when DB2 and IMS batch programs reconnect. Any indoubt units of work are resolved at reconnect time.

Specifying checkpoint frequency in IMS programs

A checkpoint indicates a commit point in IMS programs. You should specify checkpoint frequency in your program in a way that allows it to easily be changed, in case the frequency that you initially specify is not appropriate.

To specify checkpoint frequency in IMS programs:

1. Use a counter in your program to keep track of one of the following items:
 - Elapsed time
 - The number of root segments that your program accesses
 - The number of updates that your program performs
2. Issue a checkpoint call after a certain time interval, number of root segments, or number of updates.

Checkpoints in IMS programs:

Issuing checkpoint calls releases locked resources and establishes a place in the program from which you can restart the program. The decision about whether your program should issue checkpoints (and if so, how often) depends on your program.

Generally, the following types of programs should issue checkpoint calls:

- Multiple-mode programs
- Batch-oriented BMPs
- Nonmessage-driven Fast Path programs. (These programs can use a special Fast Path call, but they can also use symbolic checkpoint calls.)
- Most batch programs
- Programs that run in a data sharing environment. (Data sharing makes it possible for online and batch application programs in separate IMS systems, in the same or separate processors, to access databases concurrently. Issuing checkpoint calls frequently in programs that run in a data sharing environment is important, because programs in several IMS systems access the database.)

You do not need to issue checkpoints in the following types of programs:

- Single-mode programs
- Database load programs
- Programs that access the database in read-only mode (defined with the processing option GO during a PSBGEN) and are short enough to restart from the beginning
- Programs that, by their nature, must have exclusive use of the database

A CHKP call causes IMS to perform the following actions:

- Inform DB2 that the changes that your program made to the database can become permanent. DB2 makes the changes to DB2 data permanent, and IMS makes the changes to IMS data permanent.
- Send a message that contains the checkpoint identification that is given in the call to the system console operator and to the IMS master terminal operator.
- Return the next input message to the program's I/O area if the program processes input messages. In MPPs and transaction-oriented BMPs, a checkpoint call acts like a call for a new message.
- Sign on to DB2 again.

Programs that issue symbolic checkpoint calls can specify as many as seven data areas in the program that is to be restored at restart. DB2 always recovers to the last checkpoint. You must restart the program from that point.

If you use symbolic checkpoint calls, you can use a restart call (XRST) to restart a program after an abend. This call restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint call that the program issued before terminating abnormally.

Restriction: For BMP programs that process DB2 databases, you can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.

Checkpoints in MPPs and transaction-oriented BMPs

In single-mode programs, checkpoint calls and message retrieval calls (called get-unique calls) both establish commit points. The checkpoint calls retrieve input messages and take the place of get-unique calls. BMPs that access non-DL/I databases and MPPs can issue both get unique calls and checkpoint calls to establish commit points. However, message-driven BMPs must issue checkpoint calls rather than get-unique calls to establish commit points, because they can restart from a checkpoint only. If a program abends after issuing a get-unique call, IMS backs out the database updates to the most recent commit point, which is the get-unique call.

In multiple-mode BMPs and MPPs, the only commit points are the checkpoint calls that the program issues and normal program termination. If the program abends and it has not issued checkpoint calls, IMS backs out the program's database updates and cancels the messages that it has created since the beginning of the program. If the program has issued checkpoint calls, IMS backs out the program's changes and cancels the output messages it has created since the most recent checkpoint call.

Checkpoints in batch-oriented BMPs

If a batch-oriented BMP does not issue checkpoints frequently enough, IMS can abend that BMP or another application program for one of the following reasons:

- Other programs cannot get to the data that they need within a specified amount of time.

If a BMP retrieves and updates many database records between checkpoint calls, it can monopolize large portions of the databases and cause long waits for other programs that need those segments. (The exception to this situation is a BMP with a processing option of GO; IMS does not enqueue segments for programs with this processing option.) Issuing checkpoint calls releases the segments that the BMP has enqueued and makes them available to other programs.

- Not enough storage is available for the segments that the program has read and updated.

If IMS is using program isolation enqueueing, the space that is needed to enqueue information about the segments that the program has read and updated must not exceed the amount of storage that is defined for the IMS system. (The amount of storage available is specified during IMS system definition.) If a BMP enqueues too many segments, the amount of storage that is needed for the enqueued segments can exceed the amount of available storage. In that case, IMS terminates the program abnormally. You then need to increase the program's checkpoint frequency before rerunning the program.

When you issue a DL/I CHKP call from an application program that uses DB2 databases, IMS processes the CHKP call for all DL/I databases, and DB2 commits all the DB2 database resources. No checkpoint information is recorded for DB2 databases in the IMS log or the DB2 log. The application program must record relevant information about DB2 databases for a checkpoint, if necessary. One way to record such information is to put it in a data area that is included in the DL/I CHKP call.

Performance might be slowed by the commit processing that DB2 does during a DL/I CHKP call, because the program needs to re-establish position within a DB2 database. The fastest way to re-establish a position in a DB2 database is to use an index on the target table, with a key that matches one-to-one with every column in the SQL predicate.

Recovering data in IMS programs

Online IMS systems handle recovery and restart. For a batch region, the operational procedures control recovery and restart for your location.

To recover data in IMS programs:

Take one or more of the following actions depending on the type of program:

Program type	Recommended action
DL/I batch applications	Use the DL/I batch backout utility to back out DL/I changes. DB2 automatically backs out changes whenever the application program abends.
Applications that use symbolic checkpoints	Use a restart call (XRST) to restart a program after an abend. This call restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint call that the program issued before terminating abnormally.
BMP programs that access DB2 databases	Restart the program from the latest checkpoint. Restriction: You can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.
Applications that use online IMS systems	No action needed. Recovery and restart are part of the IMS system
Applications that reside in the batch region	Follow your location's operational procedures to control recovery and restart.

Undoing selected changes within a unit of work by using savepoints

Savepoints enable you to undo selected changes within a unit of work. Your application can set any number of savepoints and then specify a specific savepoint to indicate which changes to undo within the unit of work.

To undo selected changes within a unit of work by using savepoints:

1. Set any savepoints by using SQL `SAVEPOINT` statements. Savepoints set a point to which you can undo changes within a unit of work.

Consider the following abilities and restrictions when setting savepoints:

- You can set a savepoint with the same name multiple times within a unit of work. Each time that you set the savepoint, the new value of the savepoint replaces the old value.
 - If you do not want a savepoint to have different values within a unit of work, use the `UNIQUE` option in the `SAVEPOINT` statement. If an application executes a `SAVEPOINT` statement with the same name as a savepoint that was previously defined as unique, an SQL error occurs.
 - If you set a savepoint before you execute a `CONNECT` statement, the scope of that savepoint is the local site. If you set a savepoint after you execute the `CONNECT` statement, the scope of that savepoint is the site to which you are connected.
 - When savepoints are active, which they are until the unit of work completes, you cannot access remote sites by using three-part names or aliases for three-part names. You can, however, use DRDA access with explicit `CONNECT` statements.
 - You cannot use savepoints in global transactions, triggers, user-defined functions, or stored procedures that are nested within triggers or user-defined functions.
2. Specify the changes that you want to undo within a unit of work by using the SQL `ROLLBACK TO SAVEPOINT` statement.
DB2 undoes all changes since the specified savepoint. If you do not specify a savepoint name, DB2 rolls back work to the most recently created savepoint.
 3. Optional: If you no longer need a savepoint, delete it by using the SQL `RELEASE SAVEPOINT` statement.

Recommendation: If you no longer need a savepoint before the end of a transaction, release it. Otherwise, savepoints are automatically released at the end of a unit of work. Releasing savepoints is essential if you need to use three-part names to access remote locations, because you cannot perform this action while savepoints are active.

Examples

Rolling back to the most recently created savepoint: When the `ROLLBACK TO SAVEPOINT` statement is executed in the following code, DB2 rolls back work to savepoint B.

```
EXEC SQL SAVEPOINT A;  
...  
EXEC SQL SAVEPOINT B;  
...  
EXEC SQL ROLLBACK TO SAVEPOINT;
```


Setting savepoints during distributed processing: An application performs the following tasks:

1. Sets savepoint C1.
2. Does some local processing.
3. Executes a CONNECT statement to connect to a remote site.
4. Sets savepoint C2.

Because savepoint C1 is set before the application connects to a remote site, savepoint C1 is known only at the local site. However, because savepoint C2 is set after the application connects to the remote site, savepoint C2 is known only at the remote site.

Setting multiple savepoints with the same name: Suppose that the following actions occur within a unit of work:

1. Application A sets savepoint S.
2. Application A calls stored procedure P.
3. Stored procedure P sets savepoint S.
4. Stored procedure P executes the following statement: ROLLBACK TO SAVEPOINT S

When DB2 executes the ROLLBACK statement, DB2 rolls back work to the savepoint that was set in the stored procedure, because that value is the most recent value of savepoint S.

Related reference:

 [RELEASE SAVEPOINT \(DB2 SQL\)](#)

 [ROLLBACK \(DB2 SQL\)](#)

 [SAVEPOINT \(DB2 SQL\)](#)

Planning for recovery of table spaces that are not logged

To suppress logging, you can specify the NOT LOGGED option when you create or alter a table space. However, because logs are generally used in recovery, planning for recovery of table spaces for which changes are not logged requires some additional planning.

Although you can plan for recovery, you still need to take some corrective actions after any system failures to recover the data and fix any affected table spaces. For example, if a table space that is not logged was open for update at the time that DB2 terminates, the subsequent restart places that table space in LPL and marks it with RECOVER-pending status. You need to take corrective action to clear the RECOVER-pending status.

To plan for recovery of table spaces that are not logged:

1. Ensure that you can recover lost data by performing one of the following actions:
 - Ensure that you have a data recovery source that does not rely on a log record to re-create any lost data.
 - Limit modifications that are not logged to easily repeatable changes that can be quickly repeated.
2. Avoid placing a table space that is not logged in a RECOVER-pending status. The following actions place a table space in RECOVER-pending status:
 - Issuing a ROLLBACK statement or ROLLBACK TO SAVEPOINT statement after modifying a table in a table space that is not logged.


- Causing duplicate keys or referential integrity violations when you modify a table space that is not logged.

If the table space is placed in RECOVER-pending status, it is unavailable until you manually fix it.


3. For table spaces that are not logged and have associated LOB or XML table spaces, take image copies as a recovery set.

This action ensures that the base table space and all the associated LOB or XML table spaces are copied at the same point in time. A subsequent RECOVER TO LASTCOPY operation for the entire set results in consistent data across the base table space and all of the associated LOB and XML table spaces.

Related tasks:

 [Clearing the RECOVER-pending status \(DB2 Administration Guide\)](#)

Related reference:

 [RECOVER \(DB2 Utilities\)](#)

Designing your application to access distributed data

You can design applications that access data on another database management system (DBMS) other than your local system. You should consider the limitations and recommendations for such programs when designing them.

To design your application to access distributed data:

1. Ensure that the appropriate authorization ID has been granted authorization at the remote server to connect to that server and use resources from it.
2. If your application contains SQL statements that run at the requester, include at the requester a database request module (DBRM) that is bound directly into a package that is included in the plan's package list.
3. Copy the requester package to any remote server that is accessed by the application via a bind package copy command and include the remote packages in the application plan's package list.

Recommendation: Specify an asterisk (*) instead of a specific name in the location name of any package entry of a plan so that the plan does not have to be rebound whenever a new location is accessed by the application or a different location is to be accessed.

4. For TSO and batch applications that update data at a remote server, ensure that one of the following conditions is true:
 - No other connections exist.
 - All existing connections are to servers that are restricted to read-only operations.

Restriction: If neither of these conditions are met, the application is restricted to read-only operations.

If one of these conditions is met, and if the first connection in a logical unit of work is to a server that supports two-phase commit, that server and all servers that support two-phase commit can update data. However, if the first connection is to a server that does not support two-phase commit, only that server is allowed to update data.

5. For programs that access at least one restricted system, ensure that your program does not violate any of the limitations for accessing restricted systems. A *restricted system* is a DBMS that does not implement two-phase commit processing.

Accessing restricted systems has the following limitations:

- For programs that access CICS or IMS, you cannot update data on restricted systems.
- Within a unit of work, you cannot update a restricted system after updating a non-restricted system.
- Within a unit of work, if you update a restricted system, you cannot update any other systems.

If you are accessing a mixture of systems, some of which might be restricted, you can perform the following actions:

- Read from any of the systems at any time.
- Update any one system many times in one unit of work.
- Update many systems, including CICS or IMS, in one unit of work, provided that none of them is a restricted system. If the first system you update in a unit of work is not restricted, any attempt to update a restricted system in that unit of work returns an error.
- Update one restricted system in a unit of work, provided that you do not try to update any other system in the same unit of work. If the first system you update in a unit of work is restricted, any attempt to update any other system in that unit of work returns an error.

Related concepts:

 Phase 6: Accessing data at a remote site (DB2 Installation and Migration)

Related tasks:

 Improving performance for applications that access distributed data (DB2 Performance)

Remote servers and distributed data

Distributed data is data that resides on a database management system (DBMS) other than your local system. Your local DBMS is the one on which you bind your application plan. All other DBMSs are remote.

If you are requesting services from a remote DBMS, that DBMS is a server, and your local system is a requester or client.

Your application can be connected to many DBMSs at one time; the one that is currently performing work is the *current server*. When the local system is performing work, it also is called the current server.

A remote server can be physically remote, or it can be another subsystem of the same operating system that your local DBMS runs under. A remote server might be an instance of DB2 for z/OS, or it might be an instance of one of another product.

A DBMS, whether local or remote, is known to your DB2 system by its location name. The location name of a remote DBMS is recorded in the communications database.

Related tasks:

 Choosing names for the local subsystem (DB2 Installation and Migration)

Preparing for coordinated updates to two or more data sources

Two or more updates are coordinated if they must all commit or all roll back in the same unit of work.

This situation is common in banking. Suppose that an amount is subtracted from one account and added to another. The two actions must either both commit or both roll back at the end of the unit of work.

To prepare for coordinated updates to two or more data sources:

Ensure that all systems that your program accesses implement two-phase commit processing. This processing ensures that updates to two or more DBMSs are coordinated automatically.

For example, DB2 and IMS, and DB2 and CICS, jointly implement a two-phase commit process. You can update an IMS database and a DB2 table in the same unit of work. If a system or communication failure occurs between committing the work on IMS and on DB2, the two programs restore the two systems to a consistent point when activity resumes.

You cannot do true coordinated updates within a DBMS that does not implement two-phase commit processing, because DB2 prevents you from updating such a DBMS and any other system within the same unit of work. In this context, update includes the statements INSERT, UPDATE, MERGE, DELETE, CREATE, ALTER, DROP, GRANT, REVOKE, RENAME, COMMENT, and LABEL.

However, if you cannot implement two-phase commit processing on all systems that your program accesses, you can simulate the effect of coordinated updates by performing the following actions:

1. Update one system and commit that work.
2. Update the second system and commit its work.
3. Ensure that your program has code to undo the first update if a failure occurs after the first update is committed and before the second update is committed. No automatic provision exists for bringing the two systems back to a consistent point.

Related concepts:

 [Two-phase commit process \(DB2 Administration Guide\)](#)

Forcing restricted system rules in your program

A *restricted system* is a DBMS that does not implement two-phase commit processing. These systems have a number of update restrictions. You can restrict your program completely to the rules for these restricted systems, regardless of whether the program is accessing restricted systems or non-restricted systems.

Accessing restricted systems has the following limitations:

- For programs that access CICS or IMS, you cannot update data on restricted systems.
- Within a unit of work, you cannot update a restricted system after updating a non-restricted system.
- Within a unit of work, if you update a restricted system, you cannot update any other systems.

To force restricted system rules in your program:

When you prepare your program, specify the SQL processing option CONNECT(1). This option applies type 1 CONNECT statement rules.

Restriction: Do not use packages that are precompiled with the CONNECT(1) option and packages that are precompiled with the CONNECT(2) option in the same package list. The first CONNECT statement that is executed by your program determines which rules are in effect for the entire execution: type 1 or type 2. If your program attempts to execute a later CONNECT statement that is precompiled with the other type, DB2 returns an error.

Related concepts:

“Options for SQL statement processing” on page 932

Creating a feed in IBM Mashup Center with data from a DB2 for z/OS server

You can create enterprise database feeds based on data from a DB2 for z/OS server. A *feed* is data that is provided in a format that facilitates frequent content updates.

Introductory concepts:

IBM Mashup Center (Introduction to DB2 for z/OS)

- Install Mashup Center on a computer that is accessible from the DB2 for z/OS server.
- Configure the Mashup Center server for DB2 for z/OS.

To create a feed based on DB2 for z/OS data:

1. In the MashupHub component of Mashup Center, click **Create > New Feed** and select the Enterprise Database (JDBC) feed generator.
2. In the SQL Query Builder window, create the SQL query for the feed. The SQL parameters become the feed parameters. Parameters in the format ':arg' are treated as string parameters. Parameters in the format :arg are treated as numeric parameters. Supported SQL statements are SELECT, INSERT, UPDATE, and DELETE.
3. Save the feed and click **View Feed in Browser** to execute the SQL statement.

After you create the feed, you can do one or more of the following actions:

- Create more feeds based on DB2 for z/OS data and then mix the results of the queries in a data mashup. A *data mashup* is a feed that you create by applying operators and functions to filter and restructure the source data. Use the data mashup builder in MashupHub to create a data mashup.
- Add the feed to the Lotus® mashup builder. When you add a feed to the Mashup builder, the feed is added as a widget. A *widget* is a small application or piece of dynamic content that can be easily placed on a web page. Mashable widgets pass events so that they can be wired together to create something new.
- Add the feed to another application by using the "Add to" action.
- Share the feed in the Mashup Center community catalog so that other users can include it in their mashups. Users can tag and rate catalog objects to help others find the information that they need quickly.

Related reference:

 [Lotus Greenhouse](#)

- | [☞ IBM Mashup Center developerWorks forum](#)
- | [☞ IBM Mashup Center v2.x Information Center](#)
- | [☞ IBM Mashup Center v3.x Information Center](#)
- | [☞ Configuring MashupHub for enterprise database feeds](#)
- | [☞ IBM Mashup Center wiki](#)

Chapter 2. Connecting to DB2 from your application program

Application programs communicate with DB2 through an attachment facility. You must invoke an attachment facility, either implicitly or explicitly, before your program can interact with DB2.

You can use the following attachment facilities in a z/OS environment:

CICS attachment facility

Use this facility to access DB2 from CICS application programs.

IMS attachment facility

Use this facility to access DB2 from IMS application programs.

Time Sharing Option (TSO) attachment facility

Use this facility in a TSO or batch environment to communicate to a local DB2 subsystem. This facility invokes the DSN command processor.

Call attachment facility (CAF)

Use this facility as an alternative to the TSO attachment facility when your application needs tight control over the session environment.

Resource Recovery Services attachment facility (RRSAF)

Use this facility for stored procedures that run in a WLM-established address space or as an alternative to the CAF. RRSAF provides support for z/OS RRS as the recovery coordinator and supports other capabilities not present in CAF

For distributed applications, use the distributed data facility (DDF).

Requirement: Ensure that any application that requests DB2 services satisfies the following environment characteristics, regardless of the attachment facility that you use:

- The application must be running in TCB mode. SRB mode is not supported.
- An application task cannot have any Enabled Unlocked Task (EUT) functional recovery routines (FRRs) active when requesting DB2 services. If an EUT FRR is active, the DB2 functional recovery can fail, and your application can receive some unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. Specifically, the following requirements exist:
 - An application must not use CAF or RRSAF in an CICS or IMS address space.
 - An application that runs in an address space that has a CAF connection to DB2 cannot connect to DB2 by using RRSAF.
 - An application that runs in an address space that has an RRSAF connection to DB2 cannot connect to DB2 by using CAF.
 - An application cannot invoke the z/OS AXSET macro after executing the CAF CONNECT call and before executing the CAF DISCONNECT call.
- One attachment facility cannot start another. For example, your CAF or RRSAF application cannot use DSN, and a DSN RUN subcommand cannot call your CAF or RRSAF application.

- The language interface modules for CAF and RRSAF, DSNALI and DSNRLI, are shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load CAF or RRSAF below the 16-MB line, you must link-edit DSNALI or DSNRLI again.

Related concepts:

- ☞ DB2 attachment facilities (Introduction to DB2 for z/OS)
- ☞ Distributed data facility (Introduction to DB2 for z/OS)

Invoking the call attachment facility

Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to DB2. Applications that use CAF can explicitly control the state of their connections to DB2 by using connection functions that CAF supplies.

Before you can invoke CAF, perform the following actions:

- Ensure that the CAF language interface (DSNALI) is available.
- Ensure that your application satisfies the requirements for programs that access CAF.
- Ensure that your application satisfies the general environment characteristics for connecting to DB2.
- Ensure that you are familiar with the following z/OS concepts and facilities:
 - The CALL macro and standard module linkage conventions
 - Program addressing and residency options (AMODE and RMODE)
 - Creating and controlling tasks; multitasking
 - Functional recovery facilities such as ESTAE, ESTAI, and FRRs
 - Asynchronous events and TSO attention exits (STAX)
 - Synchronization techniques such as WAIT/POST.

Applications that use CAF can be written in assembler language, C, COBOL, Fortran, and PL/I. When choosing a language to code your application in, consider the following restrictions:

- If you need to use z/OS macros (ATTACH, WAIT, POST, and so on), use a programming language that supports them or embed them in modules that are written in assembler language.
- The CAF TRANSLATE function is not available in Fortran. To use this function, code it in a routine that is written in another language, and then call that routine from Fortran.

Recommendations: For IMS and DSN applications, consider the following recommendations:

- For IMS batch applications, do not use CAF. Instead use the DB2 DL/I batch support. Although it is possible for IMS batch applications to access DB2 databases through CAF, that method does not coordinate the commitment of work between the IMS and DB2 systems.
- For DSN applications, do not use CAF unless you provide an application controller to manage the DSN application and replace any needed DSN functions. You might also have to change the application to communicate connection failures to the controller correctly. Running DSN applications with CAF is not advantageous, and the loss of DSN services can affect how well your program runs.

To invoke CAF:

Perform one of the following actions:

- Explicitly invoke CAF by including in your program CALL DSNALI statements with the appropriate options.

The first option is a CAF connection function, which describes the action that you want CAF to take. The effect of any function depends in part on what functions the program has already run.

Requirement: For C and PL/I applications, you must also include in your program the compiler directives that are listed in the following table, because DSNALI is an assembler language program.

Table 7. Compiler directives to include in C and PL/I applications that contain CALL DSNALI statements

Language	Compiler directive to include
C	#pragma linkage(dsnali, OS)
C++	extern "OS" { int DSNALI(char * functn, ...); } }
PL/I	DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE;

- Implicitly invoke CAF by including SQL statements or IFI calls in your program just as you would in any program. The CAF facility establishes the connections to DB2 with the default values for the subsystem name and plan name.

Restriction: If your program can make its first SQL call from different modules with different DBRMs, you cannot use a default plan name and thus, you cannot implicitly invoke CAF. Instead, you must explicitly invoke CAF by using the OPEN function.

Requirement: If your application includes both SQL and IFI calls, you must issue at least one SQL call before you issue any IFI calls. This action ensures that your application uses the correct plan.

Although doing so is not recommended, you can run existing DSN applications with CAF by allowing them to make implicit connections to DB2. For DB2 to make an implicit connection successfully, the plan name for the application must be the same as the member name of the database request module (DBRM) that DB2 produced when you precompiled the source program that contains the first SQL call. You must also substitute the DSNALI language interface module for the TSO language interface module, DSNELI.

If you do not specify the return code and reason code parameters in your CAF calls or you invoked CAF implicitly, CAF puts a return code in register 15 and a reason code in register 0.

To determine if an implicit connection was successful, the application program should examine the return and reason codes immediately after the first executable SQL statement in the application program by performing one of the following actions:

- Examining registers 0 and 15 directly.
- Examining the SQLCA, and if the SQLCODE is -991, obtain the return and reason code from the message text. The return code is the first token, and the reason code is the second token.

If the implicit connection was successful, the application can examine the SQLCODE for the first, and subsequent, SQL statements.

Examples

Example of a CAF configuration: The following figure shows an conceptual example of invoking and using CAF. The application contains statements to load DSNALI, DSNHLI2, and DSNWLI2. The application accesses DB2 by using the CAF Language Interface. It calls DSNALI to handle CAF requests, DSNWLI to handle IFI calls, and DSNHLI to handle SQL calls.

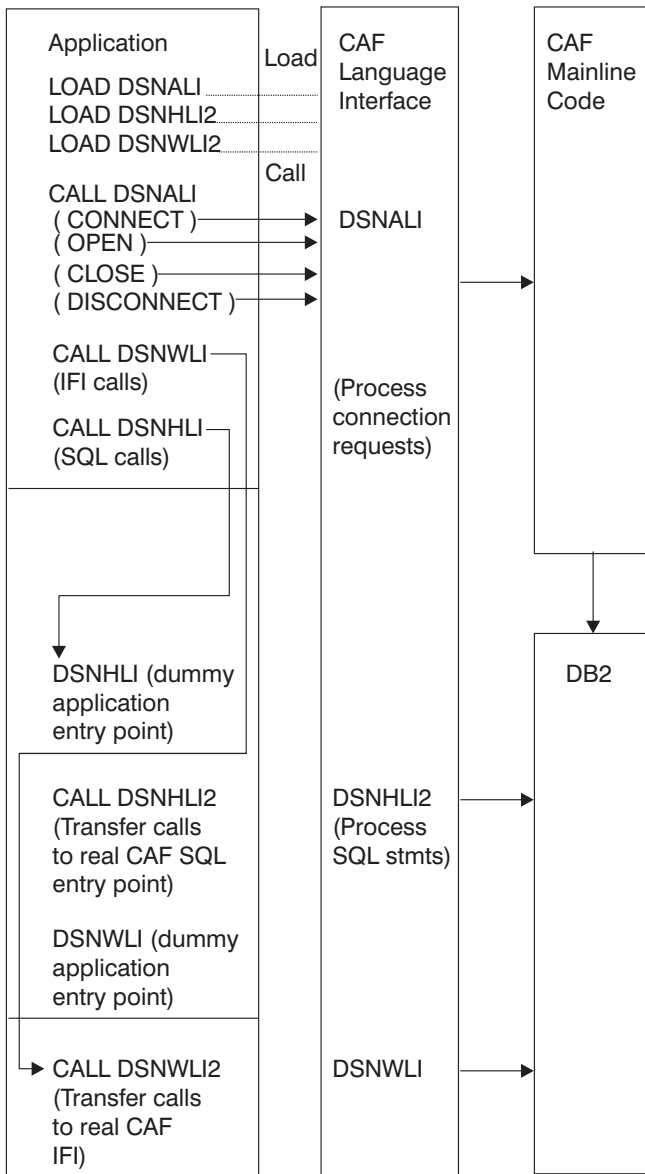


Figure 1. Sample call attachment facility configuration

Sample programs that use CAF: You can find a sample assembler program (DSN8CA) and a sample COBOL program (DSN8CC) that use the CAF in library *prefix.SDSNSAMP*. A PL/I application (DSN8SPM) calls DSN8CA, and a COBOL application (DSN8SCM) calls DSN8CC.

Related concepts:

“DB2 sample applications” on page 1092

Related reference:

“CAF connection functions” on page 86

Call attachment facility

An attachment facility enables programs to communicate with DB2. The call attachment facility (CAF) provides such a connection for programs that run in z/OS batch, TSO foreground, and TSO background. The CAF needs tight control over the session environment.

A program that uses CAF can perform the following actions:

- Access DB2 from z/OS address spaces where TSO, IMS, or CICS do not exist.
- Access DB2 from multiple z/OS tasks in an address space.
- Access the DB2 IFI.
- Run when DB2 is down.

Restriction: The application cannot run SQL when DB2 is down.

- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor or of any DB2 code.
- Run above or below the 16-MB line. (The CAF code resides below the line.)
- Establish an explicit connection to DB2, through a CALL interface, with control over the exact state of the connection.
- Establish an implicit connection to DB2, by using SQL statements or IFI calls without first calling CAF, with a default plan name and subsystem identifier.
- Verify that the application is using the correct release of DB2.
- Supply event control blocks (ECBs), for DB2 to post, that signal startup or termination.
- Intercept return codes, reason codes, and abend codes from DB2 and translate them into messages.

Any task in an address space can establish a connection to DB2 through CAF. Only one connection can exist for each task control block (TCB). A DB2 service request that is issued by a program that is running under a given task is associated with that task's connection to DB2. The service request operates independently of any DB2 activity under any other task.

Each connected task can run a plan. Multiple tasks in a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without fully breaking its connection to DB2.

CAF does not generate task structures.

When you design your application, consider that using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention.

A tracing facility provides diagnostic messages that aid in debugging programs and diagnosing errors in the CAF code. In particular, attempts to use CAF incorrectly cause error messages in the trace stream.

Restriction: CAF does not provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines and not Enabled Unlocked Task (EUT) FRR routines.

Properties of CAF connections

Call attachment facility (CAF) enables programs to communicate with DB2.

The connection that CAF makes with DB2 has the basic properties that are listed in the following table.

Table 8. Properties of CAF connections

Property	Value	Comments
Connection name	DB2CALL	You can use the DISPLAY THREAD command to list CAF applications that have the connection name DB2CALL.
Connection type	BATCH	BATCH connections use a single phase commit process that is coordinated by DB2. Application programs can also control when statements are committed by using the SQL COMMIT and ROLLBACK statements.
Authorization IDs	Authorization IDs that are associated with the address space	DB2 establishes authorization IDs for each task's connection when it processes that connection. For the BATCH connection type, DB2 creates a list of authorization IDs based on the authorization ID that is associated with the address space. This list is the same for every task. A location can provide a DB2 connection authorization exit routine to change the list of IDs.
Scope	CAF processes connections as if each task is entirely isolated. When a task requests a function, the CAF passes the functions to DB2 and is unaware of the connection status of other tasks in the address space. However, the application program and the DB2 subsystem are aware of the connection status of multiple tasks in an address space.	none

If a connected task terminates normally before the CLOSE function deallocates the plan, DB2 commits any database changes that the thread made since the last commit point. If a connected task abends before the CLOSE function deallocates

the plan, DB2 rolls back any database changes since the last commit point. In either case, DB2 deallocates the plan, if necessary, and terminates the task's connection before it allows the task to terminate.

If DB2 abnormally terminates while an application is running, the application is rolled back to the last commit point. If DB2 terminates while processing a commit request, DB2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when DB2 terminates.

Related concepts:

 [Connection routines and sign-on routines \(Managing Security\)](#)

Attention exit routines for CAF

An attention exit routine enables you to regain control from DB2 during long-running or erroneous requests. Call attachment facility (CAF) has no attention exit routines, but you can provide your own if necessary.

An attention exit routine works by detaching the TCB that is currently waiting on an SQL or IFI request to complete. After the TCB is detached, DB2 detects the resulting abend and performs termination processing for that task. The termination processing includes any necessary rollback of transactions.

You can provide your own attention exit routines. However, your routine might not get control if you request attention while DB2 code is running, because DB2 uses enabled unlocked task (EUT) functional recovery routines (FRRs).

Recovery routines for CAF

You can use abend recovery routines and functional recovery routines (FRRs) to handle unexpected errors. An abend recovery routine controls what happens when an abend occurs while DB2 has control. A functional recovery routine can obtain information about and recover from program errors.

The CAF has no abend recovery routines, but you can provide your own. Any abend recovery routines that you provide must use tracking indicators to determine if an abend occurred during DB2 processing. If an abend occurs while DB2 has control, the recovery routine can take one of the following actions:

- Allow task termination to complete. Do not try the program again. DB2 detects task termination and terminates the thread with the ABRT parameter. You lose all database changes back to the last sync point or commit point.
This action is the only action that you can take for abends that are caused by the CANCEL command or by DETACH. You cannot use additional SQL statements. If you attempt to execute another SQL statement from the application program or its recovery routine, you receive a return code of +256 and a reason code of X'00F30083'.
- In an ESTAE routine, issue a CLOSE function call with the ABRT parameter followed by a DISCONNECT function call. The ESTAE exit routine can try again so that you do not need to reinstate the application task.

FRRs must comply with the following requirements and restrictions:

- You can use only enabled unlocked task (EUT) FRRs in your routines that call DB2. The standard z/OS functional recovery routines (FRRs) apply to only code that runs in service request block (SRB) mode, and DB2 does not support calls from SRB mode routines.
- Do not have an EUT FRR active when using CAF, processing SQL requests, or calling IFI. With z/OS, if you have an active EUT FRR, all DB2 requests fail,

including the initial CONNECT or OPEN request. The requests fail because DB2 always creates an ARR-type ESTAE, and z/OS does not allow the creation of ARR-type ESTAEs when an FRR is active.

- An EUT FRR cannot retry failing DB2 requests. An EUT FRR retry bypasses ESTAE routines from DB2. The next DB2 request of any type, including a DISCONNECT request, fails with a return code of +256 and a reason code of X'00F30050'.

Making the CAF language interface (DSNALI) available

Before you can invoke the call attachment facility (CAF), you must first make DSNALI available.

Part of CAF is a DB2 load module, DSNALI, which is also known as the CAF language interface. DSNALI has the alias names DSNHLI2 and DSNWLI2. The module has five entry points: DSNALI, DSNHLI, DSNHLI2, DSNWLI, and DSNWLI2. These entry points serve the following functions:

- Entry point DSNALI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLI2 handle SQL calls. Use DSNHLI if your application program link-edits DSNALI. Use DSNHLI2 if your application program loads DSNALI.
- DSNWLI and DSNWLI2 handle IFI calls. Use DSNWLI if your application program link-edits DSNALI. Use DSNWLI2 if your application program loads DSNALI.

To make DSNALI available:

1. Decide which of the following methods you want to use to make DSNALI available:
 - Explicitly issuing LOAD requests when your program runs.
By explicitly loading the DSNALI module, you beneficially isolate the maintenance of your application from future IBM maintenance to the language interface. If the language interface changes, the change will probably not affect your load module.
 - Including the DSNALI module in your load module when you link-edit your program.

If you do not need explicit calls to DSNALI for CAF functions, link-editing DSNALI into your load module has some advantages. When you include DSNALI during the link-edit, you do not need to code a dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNALI contains an entry point for DSNHLI, which is identical to DSNHLI2, and an entry point DSNWLI, which is identical to DSNWLI2.

A disadvantage to link-editing DSNALI into your load module is that any IBM maintenance to DSNALI requires a new link-edit of your load module.

Alternatively, if using explicit connections via CALL DSNALI, you can link-edit your program with DSNULI, the Universal Language Interface.

2. Depending on the method that you chose in step 1, perform one of the following actions:
 - **If you want to explicitly issue LOAD requests when your program runs:**
In your program, issue z/OS LOAD service requests for entry points DSNALI and DSNHLI2. If you use IFI services, you must also load DSNWLI2. The entry point addresses that LOAD returns are saved for later use with the CALL macro. Indicate to DB2 which entry point to use in one of the following two ways:

- Specify the precompiler option ATTACH(CAF).
This option causes DB2 to generate calls that specify entry point DSNHLI2.

Restriction: You cannot use this option if your application is written in Fortran.

- Code a dummy entry point named DSNHLI within your load module.
If you do not specify the precompiler option ATTACH, the DB2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know about and is independent of the different DB2 attachment facilities. When the calls generated by the DB2 precompiler pass control to DSNHLI, your code that corresponds to the dummy entry point must preserve the option list that was passed in R1 and specify the same option list when it calls DSNHLI2.
- **If you want to include the DSNALI module in your load module when you link-edit your program:**

Include DSNALI in your load module during a link-edit step. The module must be in a load module library, which is included either in the SYSLIB concatenation or another INCLUDE library that is defined in the linkage editor JCL. Because all language interface modules contain an entry point declaration for DSNHLI, the linkage editor JCL must contain an INCLUDE linkage editor control statement for DSNALI; for example, INCLUDE SYSLIB(DSNALI). By coding these options, you avoid inadvertently picking up the wrong language interface module.

Related concepts:

- “LOB file reference variables” on page 767
- “Examples of invoking CAF” on page 100
- “Universal language interface” on page 153

Related tasks:

- “Link-editing an application with DSNULI” on page 155
- “Saving storage when manipulating LOBs by using LOB locators” on page 763

Requirements for programs that use CAF

The call attachment facility (CAF) enables programs to communicate with DB2. Before you invoke CAF in your program, ensure that your program satisfies any requirements for using CAF.

When you write programs that use CAF, ensure that they meet the following requirements:

- The program accounts for the size of the CAF code. The CAF code requires about 16 KB of virtual storage per address space and an additional 10 KB for each TCB that uses CAF.
- If your local environment intercepts and replaces the z/OS LOAD SVC that CAF uses, you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard z/OS LOAD macro. CAF uses z/OS SVC LOAD to load two modules as part of the initialization after your first service request. Both modules are loaded into fetch-protected storage that has the job-step protection key.
- If you use CAF from IMS batch, you must write data to only one system in any one unit of work. If you write to both systems within the same unit, a system failure can leave the two databases inconsistent with no possibility of automatic

recovery. To end a unit of work in DB2, execute the SQL COMMIT statement. To end a unit of work in IMS, issue the SYNCPOINT command.

You can prepare application programs to run in CAF similar to how you prepare applications to run in other environments, such as CICS, IMS, and TSO. You can prepare a CAF application either in the batch environment or by using the DB2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST.

Related tasks:

Chapter 17, "Preparing an application to run on DB2 for z/OS," on page 915

How CAF modifies the content of registers

If you do not specify the return code and reason code parameters in your CAF function calls or if you invoke CAF implicitly, CAF puts a return code in register 15 and a reason code in register 0. The contents of registers 2 through 14 are preserved across calls.

The following table lists the standard calling conventions for registers R1, R13, R14, and R15.

Table 9. Standard usage of registers R1, R13, R14, and R15

Register	Usage
R1	CALL DSNALI parameter list pointer
R13	Address of caller's save area
R14	Caller's return address
R15	CAF entry point address

Your CAF program should respect these register conventions.

CAF also supports high-level languages that cannot examine the contents of individual registers.

Related concepts:

"CALL DSNALI statement parameter list" on page 83

Implicit connections to CAF

If the CAF language interface (DSNALI) is available and you do not explicitly specify CALL DSNALI statements in your application, CAF initiates implicit CONNECT and OPEN requests to DB2. These requests are subject to the same DB2 return codes and reason codes as explicitly specified requests.

Implicit connections use the following defaults:

Subsystem name

The default name that is specified in the module DSNHDECP. CAF uses the installation default DSNHDECP, unless your own DSNHDECP module is in a library in a STEPLIB statement of a JOBLIB concatenation or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

Implicit connections to CAF always use DSNHDECP as the user-specified application defaults module.

Be certain that you know what the default name is and that it names the specific DB2 subsystem you want to use.

Plan name

The member name of the database request module (DBRM) that DB2 produced when you precompiled the source program that contains the first SQL call.

Different types of implicit connections exist. The simplest is for an application to call neither the CONNECT nor OPEN functions. You can also use the CONNECT function only or the OPEN function only. Each of these calls implicitly connects your application to DB2. To terminate an implicit connection, you must use the proper calls.

Related concepts:

“Summary of CAF behavior” on page 85

CALL DSNALI statement parameter list

The CALL DSNALI statement explicitly invokes CAF. When you include CALL DSNALI statements in your program, you must specify all parameters that come before the return code parameter.

For CALL DSNALI statements, use a standard z/OS CALL parameter list. Register 1 points to a list of fullword addresses that point to the actual parameters. The last address must contain a 1 in the high-order bit.

In CALL DSNALI statements, you cannot omit any of parameters that come before the return code parameter by coding zeros or blanks. No defaults exist for those parameters for explicit connection requests. Defaults are provided for only implicit connections. All parameters starting with the return code parameter are optional.

When you want to use the default value for a parameter but specify subsequent parameters, code the CALL DSNALI statement as follows:

- For C-language, when you code CALL DSNALI statements in C, you need to specify the address of every required parameter, using the “address of” operator (&), and not the parameter itself. For example, to pass the *startecb* parameter on CONNECT, specify the address of the 4-byte integer (&secb).

```
functn char[13] = "CONNECT    ";
ssid   char[ 5] = "DB2A";
int    tecb    = 0;
int    secb    = 0;
ptr    ribptr;
int    retcode;
int    reascode;
ptr    eibptr;
```

```
fnret = dsnali(&functn[0], &ssid[0], &tecb, &secb, &ribptr, &retcode, &reascode,
              NULL, &eibptr);
```

- For other languages except assembler language, code zero for that parameter in the CALL DSNALI statement. For example, suppose that you are coding a CONNECT call in a COBOL program, and you want to specify all parameters except the return code parameter. You can write a statement similar to the following statement:

```
CALL 'DSNALI' USING FUNCTN SSID TECB SECB RIBPTR
      BY CONTENT ZERO BY REFERENCE REASCODE SRDURA EIBPTR.
```


- For assembler language, code a comma for that parameter in the CALL DSNALI statement. For example, to specify all optional parameters except the return code parameter write a statement similar to the following statement:
`CALL DSNALI, (FUNCTN,SSID,TERM ECB,STARTECB,RIBPTR,,REASCODE,SRDURA,EIBPTR,GRUPOVERRIDE)`

The following figure shows a sample parameter list structure for the CONNECT function.

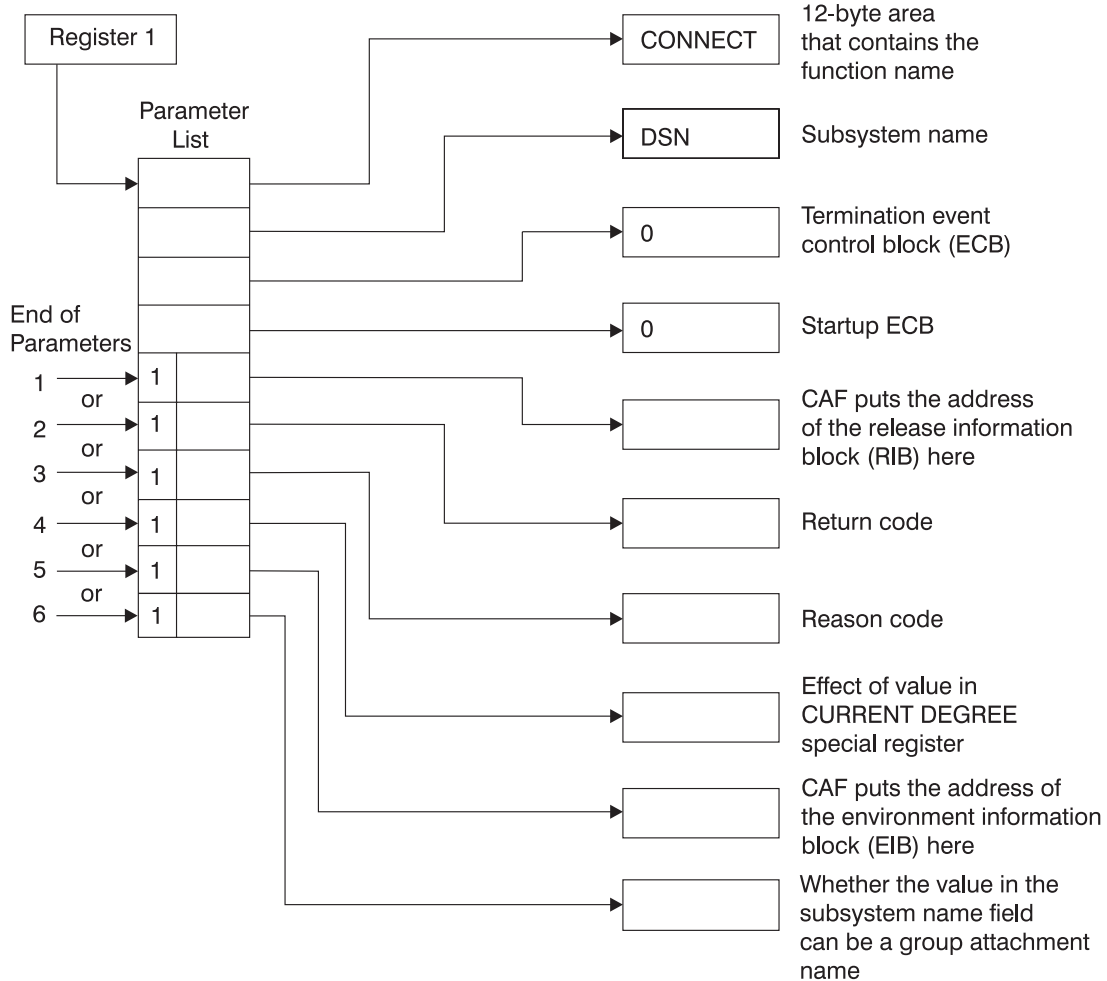


Figure 2. The parameter list for a CONNECT call

The preceding figure illustrates how you can omit parameters for the CALL DSNALI statement to control the return code and reason code fields after a CONNECT call. You can terminate the parameter list at any of the following points. These termination points apply to all CALL DSNALI statement parameter lists.

1. Terminates the parameter list without specifying the parameters *retcode*, *reascode* and *sr dura* and places the return code in register 15 and the reason code in register 0.
 Terminating the parameter list at this point ensures compatibility with CAF programs that require a return code in register 15 and a reason code in register 0.

2. Terminates the parameter list after the parameter *retcode* and places the return code in the parameter list and the reason code in register 0.
Terminating the parameter list at this point enables the application program to take action, based on the return code, without further examination of the associated reason code.
3. Terminates the parameter list after the parameter *reascode* and places the return code and the reason code in the parameter list.
Terminating the parameter list at this point provides support to high-level languages that are unable to examine the contents of individual registers.
If you code your CAF application in assembler language, you can specify the reason code parameter and omit the return code parameter.
4. Terminates the parameter list after the parameter *srdura*.
If you code your CAF application in assembler language, you can specify this parameter and omit the *retcode* and *reascode* parameters.
5. Terminates the parameter list after the parameter *eibptr*.
If you code your CAF application in assembler language, you can specify this parameter and omit the *retcode*, *reascode*, or *srdura* parameters.
6. Terminates the parameter list after the parameter *groupoverride*.
If you code your CAF application in assembler language, you can specify this parameter and omit the *retcode*, *reascode*, *srdura*, or *eibptr* parameters.

Even if you specify that the return code be placed in the parameter list, it is also placed in register 15 to accommodate high-level languages that support special return code processing.

Related concepts:

“How CAF modifies the content of registers” on page 82

Summary of CAF behavior

The effect of any CAF function depends in part on what functions the program has already run. You should plan the CAF function calls that your program makes to avoid any errors and major structural problems in your application.

The following table summarizes CAF behavior after various inputs from application programs. The top row lists the possible CAF functions that programs can call. The first column lists the task's most recent history of connection requests. For example, the value “CONNECT followed by OPEN” in the first column means that the task issued CONNECT and then OPEN with no other CAF calls in between. The intersection of a row and column shows the effect of the next call if it follows the corresponding connection history. For example, if the call is OPEN and the connection history is CONNECT, the effect is OPEN; the OPEN function is performed. If the call is SQL and the connection history is empty (meaning that the SQL call is the first CAF function the program), the effect is that implicit CONNECT and OPEN functions are performed, followed by the SQL function.

Table 10. Effects of CAF calls, as dependent on connection history

Previous function	Next function					
	CONNECT	OPEN	SQL	CLOSE	DISCONNECT	TRANSLATE
Empty: first call	CONNECT	OPEN	CONNECT, OPEN, followed by the SQL or IFI call	Error 203 ¹	Error 204 ¹	Error 205 ¹

Table 10. Effects of CAF calls, as dependent on connection history (continued)

Previous function	Next function					
	CONNECT	OPEN	SQL	CLOSE	DISCONNECT	TRANSLATE
CONNECT	Error 201 ¹	OPEN	OPEN, followed by the SQL or IFI call	Error 203 ¹	DISCONNECT	TRANSLATE
CONNECT followed by OPEN	Error 201 ¹	Error 202 ¹	The SQL or IFI call	CLOSE ²	DISCONNECT	TRANSLATE
CONNECT followed by SQL or IFI call	Error 201 ¹	Error 202 ¹	The SQL or IFI call	CLOSE ²	DISCONNECT	TRANSLATE
OPEN	Error 201 ¹	Error 202 ¹	The SQL or IFI call	CLOSE ²	Error 204 ¹	TRANSLATE
SQL or IFI call	Error 201 ¹	Error 202 ¹	The SQL or IFI call	CLOSE ²	Error 204 ¹	TRANSLATE ³

Notes:

1. An error is shown in this table as Error *nnn*. The corresponding reason code is X'00C10*nnn*'. The message number is DSNAN*nnn*I or DSNAN*nnn*E.
2. The task and address space connections remain active. If the CLOSE call fails because DB2 was down, the CAF control blocks are reset, the function produces return code 4 and reason code X'00C10824', and CAF is ready for more connection requests when DB2 is up.
3. A TRANSLATE request is accepted, but in this case it is redundant. CAF automatically issues a TRANSLATE request when an SQL or IFI request fails.

Related reference:

“CAF return codes and reason codes” on page 98

CAF connection functions

A CAF connection function specifies the action that you want CAF to take. You specify these functions when you invoke CAF through CALL DSNALI statements.

You can specify the following CAF functions in a CALL DSNALI statement:

CONNECT

Establishes the task (TCB) as a user of the named DB2 subsystem. When the first task within an address space issues a connection request, the address space is also initialized as a user of DB2.

OPEN Allocates a DB2 plan. You must allocate a plan before DB2 can process SQL statements. If you did not request the CONNECT function, the OPEN function implicitly establishes the task, and optionally the address space, as a user of DB2.

CLOSE

Commits or abnormally terminates any database changes and deallocates the plan. If the OPEN function implicitly requests the CONNECT function, the CLOSE function removes the task, and possibly the address space, as a user of DB2.

DISCONNECT

Removes the task as a user of DB2 and, if this task is the last or only task in the address space with a DB2 connection, terminates the address space connection to DB2.

TRANSLATE

Returns an SQL code and printable text that describe a DB2 hexadecimal error reason code. This information is returned to the SQLCA.

Restriction: You cannot call the TRANSLATE function from the Fortran language.

Recommendation: Because the effect of any CAF function depends on what functions the program has already run, carefully plan the calls that your program makes to these CAF connection functions. Read about the summary of CAF behavior and make these function calls accordingly.

Related concepts:

“Summary of CAF behavior” on page 85

“CALL DSNALI statement parameter list” on page 83

CONNECT function for CAF

The CAF CONNECT function initializes a connection to DB2. This function is different than the SQL CONNECT statement that accesses a remote location within DB2.

The CONNECT function establishes the caller's task as a user of DB2 services. If no other task in the address space currently holds a connection with the specified subsystem, the CONNECT function also initializes the address space for communication to the DB2 address spaces. The CONNECT function establishes the address space's cross memory authorization to DB2 and builds address space control blocks. You can issue a CONNECT request from any or all tasks in the address space, but the address space level is initialized only once when the first task connects.

Using the CONNECT function is optional. If you do not call the CONNECT function, the first request from a task, either an OPEN request or an SQL or IFI call, causes CAF to issue an implicit CONNECT request. If a task is connected implicitly, the connection to DB2 is terminated either when you call the CLOSE function or when the task terminates.

Call the CONNECT function in all of the following situations:

- You need to specify a particular subsystem name (*ssnm*) other than the default subsystem name.
- You need the value of the CURRENT DEGREE special register to last as long as the connection (*srdura*).
- You need to monitor the DB2 startup ECB (*startech*), the DB2 termination ECB (*termech*), or the DB2 release level.
- You plan to have multiple tasks in the address space open and close plans or a single task in the address space open and close plans more than once.

Establishing task and address space level connections involves significant overhead. Using the CONNECT function to establish a task connection explicitly minimizes this overhead by ensuring that the connection to DB2 remains after the CLOSE function deallocates a plan. In this case, the connection terminates only when you use the DISCONNECT function or when the task terminates.

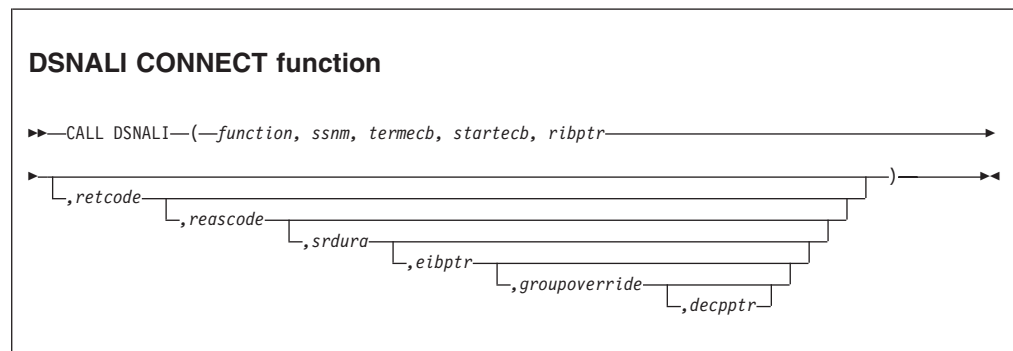
The CONNECT function also enables the caller to learn the following items:

- That the operator has issued a STOP DB2 command. When this event occurs, DB2 posts the termination ECB, *termecb*. Your application can either wait on or just look at the ECB.
- That DB2 is abnormally terminating. When this event occurs happens, DB2 posts the termination ECB, *termecb*.
- That DB2 is available again after a connection attempt that failed because DB2 was down. Your application can either wait or look at the startup ECB, *startecb*. DB2 ignores this ECB if it was active at the time of the CONNECT request.
- The current release level of DB2. To find this information, access the RIBREL field in the release information block (RIB). If RIBREL is '999', the actual version, release, and modification level of DB2 is indicated in the RIBRELX field and its subfields.

Restriction: Do not issue CONNECT requests from a TCB that already has an active DB2 connection.

Recommendation: Do not mix explicit CONNECT and OPEN requests with implicitly established connections in the same address space. Either explicitly specify which DB2 subsystem you want to use or allow all requests to use the default subsystem.

The following diagram shows the syntax for the CONNECT function.



Parameters point to the following areas:

function

A 12-byte area that contains CONNECT followed by five blanks.

ssnm

A 4-byte DB2 subsystem name or group attachment or subgroup attachment name (if used in a data sharing group) to which the connection is made.

If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

termecb

A 4-byte integer representing the application's event control block (ECB) for DB2 termination. DB2 posts this ECB when the operator enters the STOP DB2 command or when DB2 is abnormally terminating. The ECB indicates the type of termination by a POST code, as shown in the following table:

Table 11. POST codes and related termination types

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

Before you check *termcb* in your CAF application program, first check the return code and reason code from the CONNECT call to ensure that the call completed successfully.

startech

A 4-byte integer representing the application's startup ECB. If DB2 has not yet started when the application issues the call, DB2 posts the ECB when it successfully completes its startup processing. DB2 posts at most one startup ECB per address space. The ECB is the one associated with the most recent CONNECT call from that address space. Your application program must examine any nonzero CAF and DB2 reason codes before issuing a WAIT on this ECB.

If *ssnm* is a group attachment or subgroup attachment name, the first DB2 subsystem that starts on the local z/OS system and matches the specified group attachment name posts the ECB.

ribptr

A 4-byte area in which CAF places the address of the release information block (RIB) after the call. You can determine what release level of DB2 you are currently running by examining the RIBREL field. If RIBREL is '999', the actual version, release, and modification level of DB2 is indicated in the RIBRELX field and its subfields. You can determine the modification level within the release level by examining the RIBCNUMB and RIBCINFO fields. If the value in the RIBCNUMB field is greater than zero, check the RIBCINFO field for modification levels.

If the RIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *ribptr* points is below the 16-MB line.

Your program does not have to use the release information block, but it cannot omit the *ribptr* parameter.

Macro DSNDRIB maps the release information block (RIB). It can be found in *prefix.SDSNMACS(DSNDRIB)*.

retcode

A 4-byte area in which CAF places the return code.

This field is optional. If you do not specify *retcode*, CAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify *reascode*, CAF places the reason code in register 0. If you specify *reascode*, you must also specify *retcode*.

srdura

A 10-byte area that contains the string 'SRDURA(CD)'. This field is optional. If you specify *srdura*, the value in the CURRENT DEGREE special register stays in effect from the time of the CONNECT call until the time of the

DISCONNECT call. If you do not specify *srdura*, the value in the CURRENT DEGREE special register stays in effect from the time of the OPEN call until the time of the CLOSE call. If you specify this parameter in any language except assembler, you must also specify *retcode* and *reascodes*. In assembler language, you can omit these parameters by specifying commas as placeholders.

eibptr

A 4-byte area in which CAF puts the address of the environment information block (EIB). The EIB contains information that you can use if you are connecting to a DB2 subsystem that is part of a data sharing group. For example, you can determine the name of the data sharing group, the member to which you are connecting, and whether the subsystem is in new-function mode. If the DB2 subsystem that you connect to is not part of a data sharing group, the fields in the EIB that are related to data sharing are blank. If the EIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *eibptr* points is above the 16-MB line.

You can omit this parameter when you make a CONNECT call.

If you specify this parameter in any language except assembler, you must also specify *retcode*, *reascodes*, and *srdura*. In assembler language, you can omit *retcode*, *reascodes*, and *srdura* by specifying commas as placeholders.

Macro DSNDEIB maps the EIB. It can be found in *prefix.SDSNMACS(DSNDEIB)*.

groupoverride

An 8-byte area that the application provides. This parameter is optional. If you do not want group attach to be attempted, specify 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment or subgroup attachment name. If *groupoverride* is not provided, *ssnm* is used as the group attachment or subgroup attachment name if it matches a group attachment or subgroup attachment name.

If you specify this parameter in any language except assembler, you must also specify *retcode*, *reascodes*, *srdura*, and *eibptr*. In assembler language, you can omit *retcode*, *reascodes*, *srdura*, and *eibptr* by specifying commas as placeholders.

Recommendation: Avoid using the *groupoverride* parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex®. However, you should use this parameter in a data sharing environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.

decpptr

A 4-byte area in which CAF is to put the address of the DSNHDECP control block or user-specified application defaults module that was loaded by subsystem *ssnm* when that subsystem was started. This 4-byte area is a 31-bit pointer. If *ssnm* is not found, the 4-byte area is set to 0.

The area to which *decpptr* points may be above the 16-MB line.

If you specify this parameter in any language except assembler, you must also specify the *retcode*, *reascodes*, *srdura*, *eibptr*, and *groupoverride* parameters. In assembler language, you can omit the *retcode*, *reascodes*, *srdura*, *eibptr*, and *groupoverride* parameters by specifying commas as placeholders.

Example of CAF CONNECT function calls

The following table shows a CONNECT call in each language.

Table 12. Examples of CAF CONNECT function calls

Language	Call example
Assembler	CALL DSNALI, (FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA, EIBPTR, GRPOVER)
C ¹	fnret=dsnali(&functn[0],&ssid[0], &tecb, &secb,&ribptr,&retcode, &reascode, &sr dura[0], &eibptr, &grpover[0]);
COBOL	CALL 'DSNALI' USING FUNCTN SSID TERMECB STARTECB RIBPTR RETCODE REASCODE SRDURA EIBPTR GRPOVER.
Fortran	CALL DSNALI (FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA, EIBPTR,GRPOVER)
PL/I ¹	CALL DSNALI (FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA, EIBPTR,GRPOVER)

Note:

- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.


Related concepts:

“Examples of invoking CAF” on page 100

Related tasks:

“Invoking the call attachment facility” on page 74

Related reference:

 Synchronizing Tasks (WAIT, POST, and EVENTS Macros) (MVS Programming: Assembler Services Guide)

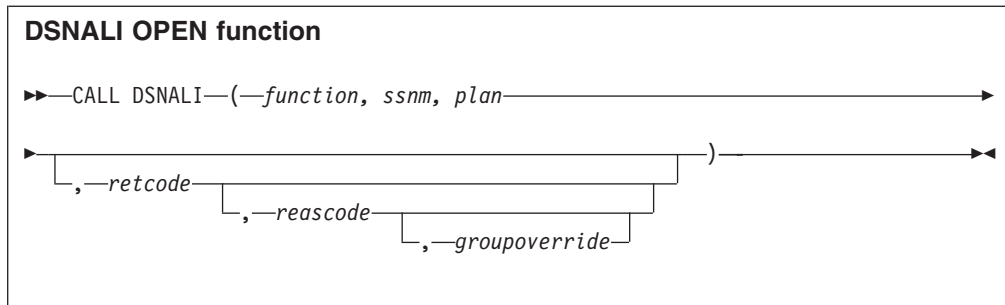
OPEN function for CAF

The OPEN function allocates DB2 resources that are needed to run the specified plan or to issue IFI requests. If the requesting task does not already have a connection to the named DB2 subsystem, the OPEN function establishes it.

Using the OPEN function is optional. If you do not call the OPEN function, the actions that the OPEN function perform occur implicitly on the first SQL or IFI call from the task.

Restriction: Do not use the OPEN function if the task already has a plan allocated.

The following diagram shows the syntax for the OPEN function.



Parameters point to the following areas:

function

A 12-byte area that contains the word OPEN followed by eight blanks.

ssnm

A 4-byte DB2 subsystem name or group attachment or subgroup attachment name (if used in a data sharing group). The OPEN function allocates the specified plan to this DB2 subsystem. Also, if the requesting task does not already have a connection to the named DB2 subsystem, the OPEN function establishes it.

You must specify the *ssnm* parameter, even if the requesting task also issues a CONNECT call. If a task issues a CONNECT call followed by an OPEN call, the subsystem names for both calls must be the same.

If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

plan

An 8-byte DB2 plan name.

retcode

A 4-byte area in which CAF places the return code.

This field is optional. If you do not specify *retcode*, CAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify *reascode*, CAF places the reason code in register 0. If you specify *reascode*, you must also specify *retcode*.

groupoverride

An 8-byte area that the application provides. This field is optional. If you do not want group attach to be attempted, specify 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment or subgroup attachment name. If you do not specify *groupoverride*, *ssnm* is used as the group attachment and subgroup attachment name if it matches a group attachment or subgroup attachment name. If you specify this parameter in any language except assembler, you must also specify *retcode* and *reascode*. In assembler language, you can omit these parameters by specifying commas as placeholders.

Recommendation: Avoid using the *groupoverride* parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex. However, you should use this parameter in a data sharing

environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.

Examples of CAF OPEN calls

The following table shows an OPEN call in each language.

Table 13. Examples of CAF OPEN calls

Language	Call example
Assembler	CALL DSNALI, (FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)
C ¹	fnret=dsnali(&functn[0],&ssid[0], &planname[0],&retcode, &reascode,&grpover[0]);
COBOL	CALL 'DSNALI' USING FUNCTN SSID PLANNAME RETCODE REASCODE GRPOVER.
Fortran	CALL DSNALI (FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)
PL/I ¹	CALL DSNALI (FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER);

Note:

- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

Related concepts:

“Implicit connections to CAF” on page 82

Related tasks:

“Invoking the call attachment facility” on page 74

CLOSE function for CAF

The CAF CLOSE function deallocates the plan that was created either explicitly by a call to the OPEN function or implicitly at the first SQL call. Optionally, the CLOSE function also disconnects the task, and possibly the address space, from DB2.

If you did not issue an explicit CONNECT call for the task, the CLOSE function deletes the task’s connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures that were created for the address space and removes the cross memory authorization.

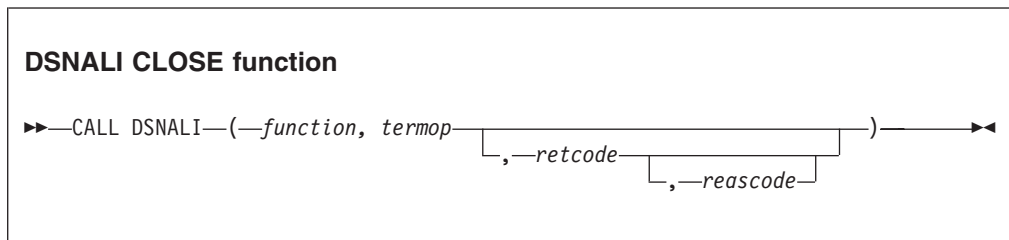
Using the CLOSE function is optional. Consider the following rules and recommendations about when to use and not use the CLOSE function:

- Do not use the CLOSE function when your current task does not have a plan allocated.
- If you want to use a new plan, you must issue an explicit CLOSE call, followed by an OPEN call with the new plan name.
- When shutting down your application you can improve the performance of this shut down by explicitly calling the CLOSE function before the task terminates. If you omit the CLOSE call, DB2 performs an implicit CLOSE. In this case, DB2 performs the same actions when your task terminates, by using the SYNC parameter if termination is normal and the ABRT parameter if termination is abnormal.
- If DB2 terminates, issue an explicit CLOSE call for any task that did not issue a CONNECT call. This action enables CAF to reset its control blocks to allow for future connections. This CLOSE call returns the reset accomplished return code

(+004) and reason code X'00C10824'. If you omit the CLOSE call in this case, when DB2 is back on line, the task's next connection request fails. You get either the message YOUR TCB DOES NOT HAVE A CONNECTION, with X'00F30018' in register 0, or the CAF error message DSNNA201I or DSNNA202I, depending on what your application tried to do. The task must then issue a CLOSE call before it can reconnect to DB2.

- A task that issued an explicit CONNECT call should issue a DISCONNECT call instead of a CLOSE call. This action causes CAF to reset its control blocks when DB2 terminates.

The following diagram shows the syntax for the CLOSE function.



Parameters point to the following areas:

function

A 12-byte area that contains the word CLOSE followed by seven blanks.

termop

A 4-byte terminate option, with one of the following values:

SYNC Specifies that DB2 is to commit any modified data.

ABRT Specifies that DB2 is to roll back data to the previous commit point.

retcode

A 4-byte area in which CAF is to place the return code.

This field is optional. If you do not specify *retcode*, CAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify *reascode*, CAF places the reason code in register 0. If you specify *reascode*, you must also specify *retcode*.

Examples of CAF CLOSE calls

The following table shows a CLOSE call in each language.

Table 14. Examples of CAF CLOSE calls

Language	Call example
Assembler	CALL DSNALI, (FUNCTN, TERMOP, RETCODE, REASCODE)
C ¹	fnret=dsnali (&functn[0], &termop[0], &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN TERMOP RETCODE REASCODE.
Fortran	CALL DSNALI (FUNCTN, TERMOP, RETCODE, REASCODE)
PL/I ¹	CALL DSNALI (FUNCTN, TERMOP, RETCODE, REASCODE);

Note:

- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

Related tasks:

“Invoking the call attachment facility” on page 74

DISCONNECT function for CAF

The CAF DISCONNECT function terminates a connection to DB2.

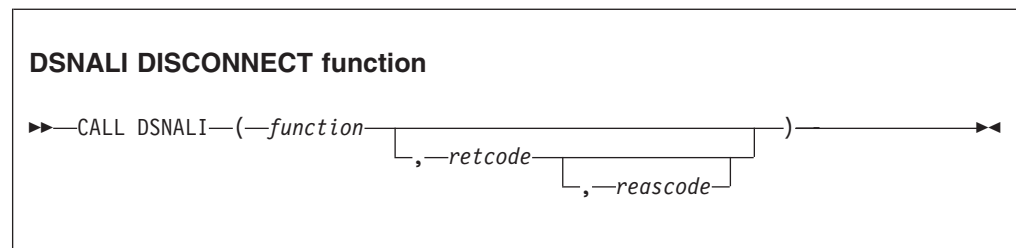
DISCONNECT removes the calling task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures that were created for the address space and removes the cross memory authorization.

If an OPEN call is in effect, which means that a plan is allocated, when the DISCONNECT call is issued, CAF issues an implicit CLOSE with the SYNC parameter.

Using the DISCONNECT function is optional. Consider the following rules and recommendations about when to use and not use the DISCONNECT function:

- Only those tasks that explicitly issued a CONNECT call can issue a DISCONNECT call. If a CONNECT call was not used, a DISCONNECT call causes an error.
- When shutting down your application you can improve the performance of this shut down by explicitly calling the DISCONNECT function before the task terminates. If you omit the DISCONNECT call, DB2 performs an implicit DISCONNECT. In this case, DB2 performs the same actions when your task terminates.
- If DB2 terminates, any task that issued a CONNECT call must issue a DISCONNECT call to reset the CAF control blocks. The DISCONNECT function returns the reset accomplished return codes and reason codes (+004 and X'00C10824'). This action ensures that future connection requests from the task work when DB2 is back on line.
- A task that did not explicitly issue a CONNECT call must issue a CLOSE call instead of a DISCONNECT call. This action resets the CAF control blocks when DB2 terminates.

The following diagram shows the syntax for the DISCONNECT function.



The single parameter points to the following area:

function

A 12-byte area that contains the word DISCONNECT followed by two blanks.

retcode

A 4-byte area in which CAF places the return code.

This field is optional. If you do not specify *retcode*, CAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify *reascode*, CAF places the reason code in register 0. If you specify *reascode*, you must also specify *retcode*.

Examples of CAF DISCONNECT calls

The following table shows a DISCONNECT call in each language.

Table 15. Examples of CAF DISCONNECT calls

Language	Call example
Assembler	CALL DSNALI(,FUNCTN,RETCODE,REASCODE)
C ¹	fnret=dsnali(&functn[0], &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN RETCODE REASCODE.
Fortran	CALL DSNALI(FUNCTN,RETCODE,REASCODE)
PL/I ¹	CALL DSNALI(FUNCTN,RETCODE,REASCODE);

Note:

- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

Related tasks:

“Invoking the call attachment facility” on page 74

TRANSLATE function for CAF

The TRANSLATE function converts a DB2 hexadecimal error reason code from a failed OPEN request into an SQL error code and printable error message text. DB2 places the information into the SQLCODE and SQLSTATE host variables or related fields of the SQLCA of the caller.

The DB2 error reason code that is converted is read from register 0. The TRANSLATE function does not change the contents of registers 0 and 15, unless the TRANSLATE request fails; in that case, register 0 is set to X'C10205' and register 15 is set to 200.

Consider the following rules and recommendations about when to use and not use the TRANSLATE function:

- You cannot call the TRANSLATE function from the Fortran language.
- The TRANSLATE function is useful only if you used an explicit CONNECT call before an OPEN request that fails. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.
- The TRANSLATE function can translate those codes that begin with X'00F3', but it does not translate CAF reason codes that begin with X'00C1'.

If you receive error reason code X'00F30040' (*resource unavailable*) after an OPEN request, the TRANSLATE function returns the name of the unavailable database object in the last 44 characters of the SQLERRM field.

Turning on a CAF trace

CAF does not capture any diagnostic trace messages unless you tell it to by turning on a trace.

To turn on a CAF trace:

Allocate a DSNTRACE data set either dynamically or by including a DSNTRACE DD statement in your JCL. CAF writes diagnostic trace messages to that data set. The trace message numbers contain the last three digits of the reason codes.

Related concepts:

“Examples of invoking CAF” on page 100

CAF return codes and reason codes

CAF provides the return codes either to the corresponding parameters that are specified in a CAF function call or, if you choose not to use those parameters, to registers 15 and 0.

When the reason code begins with X'00F3' except for X'00F30006', you can use the CAF TRANSLATE function to obtain error message text that can be printed and displayed. These reason codes are issued by the subsystem support for allied memories, a part of the DB2 subsystem support subcomponent that services all DB2 connection and work requests.

For SQL calls, CAF returns standard SQL codes in the SQLCA. CAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

The following table lists the CAF return codes and reason codes.

Table 17. CAF return codes and reason codes

Return code	Reason code	Explanation
0	X'00000000'	Successful completion.
4	X'00C10824'	CAF reset complete. CAF is ready to make a new connection.
8	X'00C10831'	Release level mismatch between DB2 and the CAF code.
200 ¹	X'00C10201'	Received a second CONNECT request from the same TCB. The first CONNECT request could have been implicit or explicit.
200 ¹	X'00C10202'	Received a second OPEN request from the same TCB. The first OPEN request could have been implicit or explicit.
200 ¹	X'00C10203'	CLOSE request issued when no active OPEN request exists.
200 ¹	X'00C10204'	DISCONNECT request issued when no active CONNECT request exists, or the AXSET macro was issued between the CONNECT request and the DISCONNECT request.
200 ¹	X'00C10205'	TRANSLATE request issued when no connection to DB2 exists.
200 ¹	X'00C10206'	Incorrect number of parameters was specified or the end-of-list bit was off.
200 ¹	X'00C10207'	Unrecognized function parameter.
200 ¹	X'00C10208'	Received requests to access two different DB2 subsystems from the same TCB.
204	²	CAF system error. Probable error in the attach or DB2.

Table 17. CAF return codes and reason codes (continued)

Return code	Reason code	Explanation
Notes:		
1.	A CAF error probably caused by errors in the parameter lists from the application programs. CAF errors do not change the current state of your connection to DB2; you can continue processing with a corrected request.	
2.	System errors cause abends. If tracing is on, a descriptive message is written to the DSNTRACE data set just before the abend.	

Sample CAF scenarios

One or more tasks can use call attachment facility (CAF) to connect to DB2. This connection can be made either implicitly or explicitly. For explicit connections, a task calls one or more of the CAF connection functions.

A single task with implicit connections

The simplest connection scenario is a single task that makes calls to DB2 without using explicit CALL DSNALI statements. The task implicitly connects to the default subsystem name and uses the default plan name.

When the task terminates, the following events occur:

- If termination was normal, any database changes are committed.
- If termination was abnormal, any database changes are rolled back.
- The active plan and all database resources are deallocated.
- The task and address space connections to DB2 are terminated.

A single task with explicit connections

The following example pseudocode illustrates a more complex scenario with a single task.

```
CONNECT
  OPEN          allocate a plan
  SQL or IFI call
  ...
  CLOSE        deallocate the current plan
  OPEN          allocate a new plan
  SQL or IFI call
  ...
  CLOSE
DISCONNECT
```

A task can have a connection to only one DB2 subsystem at any point in time. A CAF error occurs if the subsystem name in the OPEN call does not match the subsystem name in the CONNECT call. To switch to a different subsystem, the application must first disconnect from the current subsystem and then issue a connect request with a new subsystem name.

Multiple tasks

In the following scenario, multiple tasks within the address space use DB2 services. Each task must explicitly specify the same subsystem name on either the CONNECT function request or the OPEN function request. Task 1 makes no SQL or IFI calls. Its purpose is to monitor the DB2 termination and startup ECBs and to check the DB2 release level.

TASK 1	TASK 2	TASK 3	TASK n
CONNECT			
	OPEN	OPEN	OPEN
	SQL	SQL	SQL

	CLOSE	CLOSE	CLOSE
	OPEN	OPEN	OPEN
	SQL	SQL	SQL

	CLOSE	CLOSE	CLOSE
DISCONNECT			

Examples of invoking CAF

The call attachment facility (CAF) enables programs to communicate with DB2. If you explicitly invoke CAF in your program, you can use the CAF connection functions to control the state of the connection.

Example JCL for invoking CAF

The following sample JCL shows how to use CAF in a batch (non-TSO) environment. The DSNTRACE statement in this example is optional.

```
//jobname      JOB      z/OS_jobcard_information
//CAFJCL       EXEC     PGM=CAF_application_program
//STEPLIB     DD      DSN=application_load_library
//            DD      DSN=DB2_load_library

:

//SYSPRINT    DD      SYSOUT=*
//DSNTRACE    DD      SYSOUT=*
//SYSUDUMP    DD      SYSOUT=*
```

Example of assembler code that invokes CAF

The following examples show parts of a sample assembler program that uses CAF. They demonstrate the basic techniques for making CAF calls, but do not show the code and z/OS macros needed to support those calls. For example, many applications need a two-task structure so that attention-handling routines can detach connected subtasks to regain control from DB2. This structure is not shown in the following code examples. Also, these code examples assume the existence of a WRITE macro. Wherever this macro is included in the example, substitute code of your own. You must decide what you want your application to do in those situations; you probably do not want to write the error messages shown.

Example of loading and deleting the CAF language interface: The following code segment shows how an application can load entry points DSNALI and DSNHLI2 for the CAF language interface. Storing the entry points in variables LIALI and LISQL ensures that the application has to load the entry points only once. When the module is done with DB2, you should delete the entries.

```
***** GET LANGUAGE INTERFACE ENTRY ADDRESSES
      LOAD EP=DSNALI      Load the CAF service request EP
      ST  R0,LIALI        Save this for CAF service requests
      LOAD EP=DSNHLI2    Load the CAF SQL call Entry Point
      ST  R0,LISQL        Save this for SQL calls
*
*      .      Insert connection service requests and SQL calls here
*
      DELETE EP=DSNALI    Correctly maintain use count
      DELETE EP=DSNHLI2  Correctly maintain use count
```

Example of connecting to DB2 with CAF: The following example code shows how to issue explicit requests for certain actions, such as CONNECT, OPEN, CLOSE, DISCONNECT, and TRANSLATE, and uses the CHEKCODE subroutine to check the return reason codes from CAF.

```

***** CONNECT *****
      L    R15,LIALI          Get the Language Interface address
      MVC  FUNCTN,CONNECT    Get the function to call
      CALL (15),(FUNCTN,SSID,TECB,SECB,RIBPTR),VL,MF=(E,CAFCALL)
      BAL  R14,CHEKCODE      Check the return and reason codes
      CLC  CONTROL,CONTINUE  Is everything still OK
      BNE  EXIT              If CONTROL not 'CONTINUE', stop loop
      USING R8,RIB           Prepare to access the RIB
      L    R8,RIBPTR         Access RIB to get DB2 release level
      CLC  RIBREL,RIBR999    DB2 V10 or later?
      BE   USERELX          If RIBREL = '999', use RIBRELX
      WRITE 'The current DB2 release level is' RIBREL
      B    OPEN              Continue with signon
      USERELX WRITE 'The current DB2 release level is' RIBRELX

***** OPEN *****
OPEN   L    R15,LIALI          Get the Language Interface address
      MVC  FUNCTN,OPEN        Get the function to call
      CALL (15),(FUNCTN,SSID,PLAN),VL,MF=(E,CAFCALL)
      BAL  R14,CHEKCODE      Check the return and reason codes

***** SQL *****
*      Insert your SQL calls here. The DB2 Precompiler
*      generates calls to entry point DSNHLI. You should
*      specify the precompiler option ATTACH(CAF), or code
*      a dummy entry point named DSNHLI to intercept
*      all SQL calls. A dummy DSNHLI is shown below.
***** CLOSE *****
      CLC  CONTROL,CONTINUE  Is everything still OK?
      BNE  EXIT              If CONTROL not 'CONTINUE', shut down
      MVC  TRMOP,ABRT        Assume termination with ABRT parameter
      L    R4,SQLCODE        Put the SQLCODE into a register
      C    R4,CODE0          Examine the SQLCODE
      BZ   SYNCTERM          If zero, then CLOSE with SYNC parameter
      C    R4,CODE100        See if SQLCODE was 100
      BNE  DISC              If not 100, CLOSE with ABRT parameter
      SYNCTERM MVC TRMOP,SYNC Good code, terminate with SYNC parameter
DISC   DS   0H              Now build the CAF parmlist
      L    R15,LIALI          Get the Language Interface address
      MVC  FUNCTN,CLOSE      Get the function to call
      CALL (15),(FUNCTN,TRMOP),VL,MF=(E,CAFCALL)
      BAL  R14,CHEKCODE      Check the return and reason codes

***** DISCONNECT *****
      CLC  CONTROL,CONTINUE  Is everything still OK
      BNE  EXIT              If CONTROL not 'CONTINUE', stop loop
      L    R15,LIALI          Get the Language Interface address
      MVC  FUNCTN,DISCON     Get the function to call
      CALL (15),(FUNCTN),VL,MF=(E,CAFCALL)
      BAL  R14,CHEKCODE      Check the return and reason codes

```

This example code does not show a task that waits on the DB2 termination ECB. If you want such a task, you can code it by using the z/OS WAIT macro to monitor the ECB. You probably want this task to detach the sample code if the termination ECB is posted. That task can also wait on the DB2 startup ECB. This sample waits on the startup ECB at its own task level.

This example code assumes that the variables in the following table are already set:

Table 18. Variables that preceding example assembler code assumes are set

Variable	Usage
LIALI	The entry point that handles DB2 connection service requests.
LISQL	The entry point that handles SQL calls.
SSID	The DB2 subsystem identifier.
TECB	The address of the DB2 termination ECB.
SECB	The address of the DB2 startup ECB.
RIBPTR	A fullword that CAF sets to contain the RIB address.
PLAN	The plan name to use in the OPEN call.
CONTROL	This variable is used to shut down processing because of unsatisfactory return or reason codes. The CHECKCODE subroutine sets this value.
CAFCALL	List-form parameter area for the CALL macro.

Example of checking return codes and reason codes when using CAF: The following example code illustrates a way to check the return codes and the DB2 termination ECB after each connection service request and SQL call. The routine sets the variable CONTROL to control further processing within the module.

```

*****
* CHEKCODE PSEUDOCODE
*****
*IF TECB is POSTed with the ABTERM or FORCE codes
* THEN
*   CONTROL = 'SHUTDOWN'
*   WRITE 'DB2 found FORCE or ABTERM, shutting down'
* ELSE
*   SELECT (RETCODE)
*     WHEN (0) ;
*     WHEN (4) ;
*     SELECT (REASCODE)
*       WHEN ('00C10824'X)
*         CONTROL = 'RESTART'
*       OTHERWISE
*         WRITE 'Found unexpected R0 when R15 was 4'
*         CONTROL = 'SHUTDOWN'
*     END INNER-SELECT
*   WHEN (8,12)
*     SELECT (REASCODE)
*       WHEN ('00C10831'X)
*         WRITE 'Found a mismatch between DB2 and CAF release levels'
*       WHEN ('00F30002'X,
*            '00F30012'X)
*         DO
*           WRITE 'DB2 is unavailable. I'll tell you when it is up.'
*           WAIT SECB
*           WRITE 'DB2 is now available.'
*         END
*   *****
*   /* Insert tests for other DB2 connection failures here.
*   /* CAF Externals Specification lists other codes you can
*   /* receive. Handle them in whatever way is appropriate
*   /* for your application.
*   *****
*   OTHERWISE

```

```

*          WRITE 'Warning: DB2 connection failure. Cause unknown'
*          CALL DSNALI ('TRANSLATE',SQLCA) /* Fill in SQLCA */
*          WRITE SQLCODE and SQLERRM
*          END INNER-SELECT
*      WHEN (200)
*          WRITE 'CAF found user error. See DSNTRACE data set'
*      WHEN (204)
*          WRITE 'CAF system error. See DSNTRACE data set'
*      OTHERWISE
*          CONTROL = 'SHUTDOWN'
*          WRITE 'Got an unrecognized return code'
*      END MAIN SELECT
*      IF (RETCODE > 4) THEN          /* Was there a connection problem?*/
*          CONTROL = 'SHUTDOWN'
*      END CHEKCODE

*****
* Subroutine CHEKCODE checks return codes from DB2 and Call Attach.
* When CHEKCODE receives control, R13 should point to the caller's
* save area.
*****
CHEKCODE DS    0H
          STM   R14,R12,12(R13)    Prolog
          ST    R15,RETCODE        Save the return code
          ST    R0,REASCODE        Save the reason code
          LA   R15,SAVEAREA        Get save area address
          ST   R13,4(,R15)         Chain the save areas
          ST   R15,8(,R13)         Chain the save areas
          LR   R13,R15             Put save area address in R13
*      ***** HUNT FOR FORCE OR ABTERM *****
          TM   TECB,POSTBIT        See if TECB was POSTed
          BZ   DOCHECKS            Branch if TECB was not POSTed
          CLC  TECBCODE(3),QUIESCE  Is this "STOP DB2 MODE=FORCE"
          BE   DOCHECKS            If not QUIESCE, was FORCE or ABTERM
          MVC  CONTROL,SHUTDOWN    Shutdown
          WRITE 'Found found FORCE or ABTERM, shutting down'
          B    ENDCCODE            Go to the end of CHEKCODE
DOCHECKS DS    0H                Examine RETCODE and REASCODE
*      ***** HUNT FOR 0 *****
          CLC  RETCODE,ZERO        Was it a zero?
          BE   ENDCCODE            Nothing to do in CHEKCODE for zero
*      ***** HUNT FOR 4 *****
          CLC  RETCODE,FOUR        Was it a 4?
          BNE  HUNT8                If not a 4, hunt eights
          CLC  REASCODE,C10831      Was it a release level mismatch?
          BNE  HUNT824              Branch if not an 831
          WRITE 'Found a mismatch between DB2 and CAF release levels'
          B    ENDCCODE            We are done. Go to end of CHEKCODE
HUNT824  DS    0H                Now look for 'CAF reset' reason code
          CLC  REASCODE,C10824      Was it 4? Are we ready to restart?
          BNE  UNRECOG             If not 824, got unknown code
          WRITE 'CAF is now ready for more input'
          MVC  CONTROL,RESTART      Indicate that we should re-CONNECT
          B    ENDCCODE            We are done. Go to end of CHEKCODE
UNRECOG  DS    0H
          WRITE 'Got RETCODE = 4 and an unrecognized reason code'
          MVC  CONTROL,SHUTDOWN    Shutdown, serious problem
          B    ENDCCODE            We are done. Go to end of CHEKCODE
*      ***** HUNT FOR 8 *****
HUNT8    DS    0H
          CLC  RETCODE,EIGHT        Hunt return code of 8
          BE   GOT8OR12
          CLC  RETCODE,TWELVE      Hunt return code of 12
          BNE  HUNT200
GOT8OR12 DS    0H                Found return code of 8 or 12
          WRITE 'Found RETCODE of 8 or 12'
          CLC  REASCODE,F30002      Hunt for X'00F30002'
          BE   DB2DOWN

```



```

        CLC   REASCODE,F30012   Hunt for X'00F30012'
        BE   DB2DOWN
        WRITE 'DB2 connection failure with an unrecognized REASCODE'
        CLC   SQLCODE,ZERO     See if we need TRANSLATE
        BNE  A4TRANS          If not blank, skip TRANSLATE
*       ***** TRANSLATE unrecognized RETCODEs *****
        WRITE 'SQLCODE 0 but R15 not, so TRANSLATE to get SQLCODE'
        L    R15,LIALI        Get the Language Interface address
        CALL (15),(TRANSLAT,SQLCA),VL,MF=(E,CAFCALL)
        C    R0,C10205        Did the TRANSLATE work?
        BNE  A4TRANS          If not C10205, SQLERRM now filled in
        WRITE 'Not able to TRANSLATE the connection failure'
        B    ENDCCODE         Go to end of CHEKCODE
A4TRANS DS    0H             SQLERRM must be filled in to get here
*       Note: your code should probably remove the X'FF'
*       separators and format the SQLERRM feedback area.
*       Alternatively, use DB2 Sample Application DSNTIAR
*       to format a message.
        WRITE 'SQLERRM is:' SQLERRM
        B    ENDCCODE         We are done. Go to end of CHEKCODE
DB2DOWN DS    0H             Hunt return code of 200
        WRITE 'DB2 is down and I will tell you when it comes up'
        WAIT ECB=SECB        Wait for DB2 to come up
        WRITE 'DB2 is now available'
        MVC  CONTROL,RESTART  Indicate that we should re-CONNECT
        B    ENDCCODE
*       ***** HUNT FOR 200 *****
HUNT200 DS    0H             Hunt return code of 200
        CLC   RETCODE,NUM200   Hunt 200
        BNE  HUNT204
        WRITE 'CAF found user error, see DSNTRACE data set'
        B    ENDCCODE         We are done. Go to end of CHEKCODE
*       ***** HUNT FOR 204 *****
HUNT204 DS    0H             Hunt return code of 204
        CLC   RETCODE,NUM204   Hunt 204
        BNE  WASSAT           If not 204, got strange code
        WRITE 'CAF found system error, see DSNTRACE data set'
        B    ENDCCODE         We are done. Go to end of CHEKCODE
*       ***** UNRECOGNIZED RETCODE *****
WASSAT  DS    0H
        WRITE 'Got an unrecognized RETCODE'
        MVC  CONTROL,SHUTDOWN Shutdown
        BE   ENDCCODE         We are done. Go to end of CHEKCODE
ENDCCODE DS    0H             Should we shut down?
        L    R4,RETCODE        Get a copy of the RETCODE
        C    R4,FOUR           Have a look at the RETCODE
        BNH  BYEBYE           If RETCODE <= 4 then leave CHEKCODE
        MVC  CONTROL,SHUTDOWN Shutdown
BYEBYE  DS    0H             Wrap up and leave CHEKCODE
        L    R13,4(,R13)       Point to caller's save area
        RETURN (14,12)         Return to the caller

```

Example of invoking CAF when you do not specify the precompiler option ATTACH(CAF): Each of the four DB2 attachment facilities contains an entry point named DSNHLLI. When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLLI in your program. To find the correct DSNHLLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLLI that passes control to entry point DSNHLLI2 in the DSNALI module. DSNHLLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
*****
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
*****
DS      0D
DSNHLI  CSECT          Begin CSECT
        STM   R14,R12,12(R13) Prologue
        LA    R15,SAVEHLI Get save area address
        ST   R13,4(,R15) Chain the save areas
        ST   R15,8(,R13) Chain the save areas
        LR   R13,R15      Put save area address in R13
        L    R15,LISQL    Get the address of real DSNHLI
        BASSM R14,R15     Branch to DSNALI to do an SQL call
        *                DSNALI is in 31-bit mode, so use
        *                BASSM to assure that the addressing
        *                mode is preserved.
        L    R13,4(,R13) Restore R13 (caller's save area addr)
        L    R14,12(,R13) Restore R14 (return address)
        RETURN (1,12)     Restore R1-12, NOT R0 and R15 (codes)
```

Example of variable declarations when using CAF: The following example code shows declarations for some of the variables that were used in the previous subroutines.

```
***** VARIABLES *****
SECB    DS    F          DB2 Startup ECB
TECB    DS    F          DB2 Termination ECB
LIALI   DS    F          DSNALI Entry Point address
LISQL   DS    F          DSNHLI2 Entry Point address
SSID    DS    CL4       DB2 Subsystem ID. CONNECT parameter
PLAN    DS    CL8       DB2 Plan name. OPEN parameter
TRMOP   DS    CL4       CLOSE termination option (SYNC|ABRT)
FUNCTN  DS    CL12      CAF function to be called
RIBPTR  DS    F          DB2 puts Release Info Block addr here
RETCODE DS    F          Chekcode saves R15 here
REASCODE DS   F          Chekcode saves R0 here
CONTROL DS    CL8       GO, SHUTDOWN, or RESTART
SAVEAREA DS 18F        Save area for CHEKCODE
***** CONSTANTS *****
SHUTDOWN DC  CL8'SHUTDOWN' CONTROL value: Shutdown execution
RESTART  DC  CL8'RESTART ' CONTROL value: Restart execution
CONTINUE DC  CL8'CONTINUE' CONTROL value: Everything OK, cont
CODE0    DC  F'0'        SQLCODE of 0
CODE100  DC  F'100'      SQLCODE of 100
QUIESCE  DC  XL3'000008' TECB postcode: STOP DB2 MODE=QUIESCE
CONNECT  DC  CL12'CONNECT ' Name of a CAF service. Must be CL12!
OPEN     DC  CL12'OPEN   ' Name of a CAF service. Must be CL12!
CLOSE    DC  CL12'CLOSE  ' Name of a CAF service. Must be CL12!
DISCON   DC  CL12'DISCONNECT ' Name of a CAF service. Must be CL12!
TRANSLAT DC  CL12'TRANSLATE ' Name of a CAF service. Must be CL12!
SYNC     DC  CL4'SYNC'   Termination option (COMMIT)
ABRT     DC  CL4'ABRT'   Termination option (ROLLBACK)
***** RETURN CODES (R15) FROM CALL ATTACH ****
ZERO     DC  F'0'        0
FOUR     DC  F'4'        4
EIGHT    DC  F'8'        8
TWELVE   DC  F'12'      12 (Call Attach return code in R15)
NUM200   DC  F'200'     200 (User error)
NUM204   DC  F'204'     204 (Call Attach system error)
***** REASON CODES (R00) FROM CALL ATTACH ****
C10205   DC  XL4'00C10205' Call attach could not TRANSLATE
C10831   DC  XL4'00C10831' Call attach found a release mismatch
C10824   DC  XL4'00C10824' Call attach ready for more input
F30002   DC  XL4'00F30002' DB2 subsystem not up
```

```

F30011 DC XL4'00F30011' DB2 subsystem not up
F30012 DC XL4'00F30012' DB2 subsystem not up
F30025 DC XL4'00F30025' DB2 is stopping (REASCODE)
*
*      Insert more codes here as necessary for your application
*
***** SQLCA and RIB *****
EXEC SQL INCLUDE SQLCA
      DSNDRIB          Get the DB2 Release Information Block
***** CALL macro parm list *****
CAFCALL CALL  ,(*,*,*,*,*,*,*,*,*),VL,MF=L

```

Invoking the Resource Recovery Services attachment facility

The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with DB2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

Before you invoke RRSAF, perform the following actions:

- Ensure that the RRSAF language interface load module, DSNRLI, is available.
- Ensure that your application satisfies the requirements for programs that access RRSAF.
- Ensure that your application satisfies the general environment characteristics for connecting to DB2.
- Ensure that you are familiar with the following z/OS concepts and facilities:
 - The CALL macro and standard module linkage conventions
 - Program addressing and residency options (AMODE and RMODE)
 - Creating and controlling tasks; multitasking
 - Functional recovery facilities such as ESTAE, ESTAI, and FRRs
 - Synchronization techniques such as WAIT/POST
 - z/OS RRS functions, such as SRRCMIT and SRRBACK

Applications that use RRSAF can be written in assembler language, C, COBOL, Fortran, and PL/I. When choosing a language to code your application in, consider the following restrictions:

- If you use z/OS macros (ATTACH, WAIT, POST, and so on), choose a programming language that supports them.
- The RRSAF TRANSLATE function is not available in Fortran. To use this function, code it in a routine that is written in another language, and then call that routine from Fortran.

To invoke RRSAF:

1. Perform one of the following actions:
 - Explicitly invoke RRSAF by including in your program CALL DSNRLI statements with the appropriate options.

The first option is an RRSAF connection function, which describes the action that you want RRSAF to take. The effect of any function depends in part on what functions the program has already performed.

To code RRSAF functions in C, COBOL, Fortran, or PL/I, follow the individual language's rules for making calls to assembler language routines. Specify the return code and reason code parameters in the parameter list for each RRSAF call.

Requirement: For C, C++, and PL/I applications, you must also include in your program the compiler directives that are listed in the following table, because DSNRLI is an assembler language program.

Table 19. Compiler directives to include in C, C++, and PL/I applications that contain CALL DSNRLI statements

Language	Compiler directive to include
C	<code>#pragma linkage(dsnrli, OS)</code>
C++	<code>extern "OS" { int DSNRLI(char * functn, ...); }</code>
PL/I	<code>DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);</code>

- Implicitly invoke RRSF by including SQL statements or IFI calls in your program just as you would in any program. The RRSF facility establishes the connection to DB2 with the default values for the subsystem name, plan name and authorization ID.

Restriction: If your program can make its first SQL call from different modules with different DBRMs, you cannot use a default plan name and thus, you cannot implicitly invoke RRSF. Instead, you must explicitly invoke RRSF by calling the CREATE THREAD function.

Requirement: If your application includes both SQL and IFI calls, you must issue at least one SQL call before you issue any IFI calls. This action ensures that your application uses the correct plan.

2. If you implicitly invoked RRSF, determine if the implicit connection was successful by examining the return code and reason code immediately after the first executable SQL statement within the application program. Your program can check these codes by performing one of the following actions:
 - Examine registers 0 and 15 directly.
 - Examine the SQLCA, and if the SQLCODE is -981, obtain the return and reason code from the message text. The return code is the first token, and the reason code is the second token.

If the implicit connection is successful, the application can examine the SQLCODE for the first, and subsequent, SQL statements.

Example of an RRSF configuration

The following figure shows an conceptual example of invoking and using RRSF.

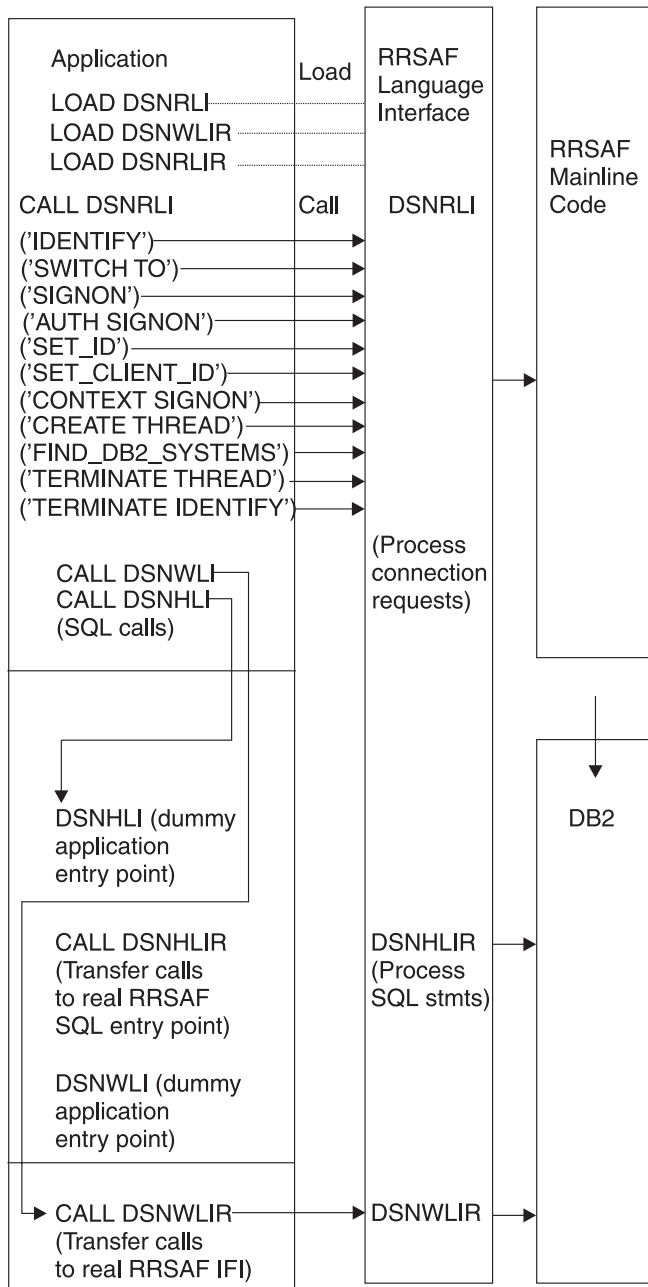


Figure 3. Sample RRSAF configuration

Resource Recovery Services attachment facility

An attachment facility enables programs to communicate with DB2. The Resource Recovery Services attachment facility (RRSAF) provides such a connection for programs that run in z/OS batch, TSO foreground, and TSO background. The RRSAF is an alternative to CAF and has more functionality.

An application program using RRSAF can perform the following actions:

- Use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls.

- Coordinate DB2 updates with updates made by all other resource managers that also use z/OS RRS in an z/OS system.
- Use the z/OS System Authorization Facility and an external security product, such as RACF, to sign on to DB2 with the authorization ID of a user.
- Sign on to DB2 using a new authorization ID and an existing connection and plan.
- Access DB2 from multiple z/OS tasks in an address space.
- Switch a DB2 thread among z/OS tasks within a single address space.
- Access the DB2 IFI.
- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor (or of any DB2 code).
- Run above or below the 16-MB line.
- Establish an explicit connection to DB2, through a call interface, with control over the exact state of the connection.
- Establish an implicit connection to DB2 (with a default subsystem identifier and a default plan name) by using SQL statements or IFI calls without first calling RRSAF.
- Supply event control blocks (ECBs), for DB2 to post, that signal start-up or termination.
- Intercept return codes, reason codes, and abend codes from DB2 and translate them into messages as required.

RRSAF uses z/OS Transaction Management and Recoverable Resource Manager Services (z/OS RRS).

Any task in an address space can establish a connection to DB2 through RRSAF. Each task control block (TCB) can have only one connection to DB2. A DB2 service request that is issued by a program that runs under a given task is associated with that task's connection to DB2. The service request operates independently of any DB2 activity under any other task.

Each connected task can run a plan. Tasks within a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without completely breaking its connection to DB2.

RRSAF does not generate task structures.

When you design your application, consider that using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention.

Restriction: RRSAF does not provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines only.

A tracing facility provides diagnostic messages that help you debug programs and diagnose errors in the RRSAF code. The trace information is available only in a SYSABEND or SYSUDUMP dump.

To commit work in RRSAF applications, use the CPIC SRRCMIT function or the DB2 COMMIT statement. To roll back work, use the CPIC SRRBACK function or the DB2 ROLLBACK statement.

Use the following guidelines to decide whether to use the DB2 statements or the CPIC functions for commit and rollback operations:

- Use DB2 COMMIT and ROLLBACK statements when all of the following conditions are true:
 - The only recoverable resource that is accessed by your application is DB2 data that is managed by a single DB2 instance.
DB2 COMMIT and ROLLBACK statements fail if your RRSAF application accesses recoverable resources other than DB2 data that is managed by a single DB2 instance.
 - The address space from which syncpoint processing is initiated is the same as the address space that is connected to DB2.
- If your application accesses other recoverable resources, or syncpoint processing and DB2 access are initiated from different address spaces, use SRRCMIT and SRRBACK.

Related reference:

 [COMMIT \(DB2 SQL\)](#)

 [ROLLBACK \(DB2 SQL\)](#)

Related information:

 [Using Protected Resources \(MVS Programming: Callable Services for High-Level Languages\)](#)

Properties of RRSAF connections

RRSAF enables programs to communicate with DB2 to process SQL statements, commands, or IFI calls.

Restriction: Do not mix RRSAF connections with other connection types in a single address space. The first connection that is made from an address space to DB2 determines the type of connection allowed.

The connection that RRSAF makes with DB2 has the basic properties that are listed in the following table.

Table 20. Properties of RRSAF connections

Property	Value	Comments
Connection name	RRSAF	You can use the DISPLAY THREAD command to list RRSAF applications that have the connection name RRSAF.
Connection type	RRSAF	None.

Table 20. Properties of RRSAF connections (continued)

Property	Value	Comments
Authorization ID	Authorization IDs that are associated with each DB2 connection	<p>A connection must have a primary ID and can have one or more secondary IDs. Those identifiers are used for the following purposes:</p> <ul style="list-style-type: none"> • Validating access to DB2 • Checking privileges on DB2 objects • Assigning ownership of DB2 objects • Identifying the user of a connection for audit, performance, and accounting traces. <p>RRSAF relies on the z/OS System Authorization Facility (SAF) and a security product, such as RACF, to verify and authorize the authorization IDs. An application that connects to DB2 through RRSAF must pass those identifiers to SAF for verification and authorization checking. RRSAF retrieves the identifiers from SAF.</p> <p>A location can provide an authorization exit routine for a DB2 connection to change the authorization IDs and to indicate whether the connection is allowed. The actual values that are assigned to the primary and secondary authorization IDs can differ from the values that are provided by a SIGNON or AUTH SIGNON request. A site's DB2 signon exit routine can access the primary and secondary authorization IDs and can modify the IDs to satisfy the site's security requirements. The exit routine can also indicate whether the signon request should be accepted.</p>

Table 20. Properties of RRSAF connections (continued)

Property	Value	Comments
Scope	RRSAF processes connections as if each task is entirely isolated. When a task requests a function, RRSAF passes the function to DB2, regardless of the connection status of other tasks in the address space. However, the application program and the DB2 subsystem have access to the connection status of multiple tasks in an address space.	None.

If an application that is connected to DB2 through RRSAF terminates normally before the `TERMINATE THREAD` or `TERMINATE IDENTIFY` functions deallocate the plan, RRS commits any changes made after the last commit point. If the application terminates abnormally before the `TERMINATE THREAD` or `TERMINATE IDENTIFY` functions deallocate the plan, z/OS RRS rolls back any changes made after the last commit point. In either case, DB2 deallocates the plan, if necessary, and terminates the application's connection.

If DB2 abends while an application is running, DB2 rolls back changes to the last commit point. If DB2 terminates while processing a commit request, DB2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when DB2 terminates.

Making the RRSAF language interface (DSNRLI) available

Before you can invoke the Resource Recovery Services attachment facility (RRSAF), you must first make available the RRSAF language interface load module, DSNRLI.

Part of RRSAF is a DB2 load module, DSNRLI, which is also known as the RRSAF language interface module. DSNRLI has the alias names DSNHLIR and DSNWLIR. The module has five entry points: DSNRLI, DSNHLI, DSNHLIR, DSNWLI, and DSNWLIR. These entry points serve the following functions:

- Entry point DSNRLI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLIR handle SQL calls. Use DSNHLI if your application program link-edits RRSAF. Use DSNHLIR if your application program loads RRSAF.
- DSNWLI and DSNWLIR handle IFI calls. Use DSNWLI if your application program link-edits RRSAF. Use DSNWLIR if your application program loads RRSAF.

To make DSNRLI available:

1. Decide which of the following methods you want to use to make DSNRLI available:
 - Explicitly issuing `LOAD` requests when your program runs.

By explicitly loading the DSNRLI module, you can isolate the maintenance of your application from future IBM maintenance to the language interface. If the language interface changes, the change will probably not affect your load module.

- Including the DSNRLI module in your load module when you link-edit your program.

A disadvantage of link-editing DSNRLI into your load module is that if IBM makes a change to DSNRLI, you must link-edit your program again.

Alternatively, if using explicit connections via CALL DSNALI, you can link-edit your program with DSNULI, the Universal Language Interface.

2. Depending on the method that you chose in step 1, perform one of the following actions:

- **If you want to explicitly issue LOAD requests when your program runs:**

In your program, issue z/OS LOAD service requests for entry points DSNRLI and DSNHLIR. If you use IFI services, you must also load DSNWLIR. Save the entry point address that LOAD returns and use it in the CALL macro.

Indicate to DB2 which entry point to use in one of the following two ways:

- Specify the precompiler option ATTACH(RRSAP).

This option causes DB2 to generate calls that specify entry point DSNHLIR.

Restriction: You cannot use this option if your application is written in Fortran.

- Code a dummy entry point named DSNHLI within your load module.

If you do not specify the precompiler option ATTACH, the DB2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know about and is independent of the different DB2 attachment facilities. When the calls that are generated by the DB2 precompiler pass control to DSNHLI, your code that corresponds to the dummy entry point must preserve the option list that is passed in register 1 and call DSNHLIR with the same option list.

- **If you want to include the DSNRLI module in your load module when you link-edit your program:**

Include DSNRLI in your load module during a link-edit step. For example, you can use a linkage editor control statement that is similar to the following statement in your JCL:

```
INCLUDE DB2LIB(DSNRLI).
```

By coding this statement, you avoid inadvertently picking up the wrong language interface module.

When you include the DSNRLI module during the link-edit, do not include a dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNRLI contains an entry point for DSNHLI, which is identical to DSNHLIR, and an entry point for DSNWLI, which is identical to DSNWLIR.

Related concepts:

“Program examples for RRSAP” on page 151

“Universal language interface” on page 153

Related tasks:

“Making the CAF language interface (DSNALI) available” on page 80

“Link-editing an application with DSNULI” on page 155

Requirements for programs that use RRSF

The Resource Recovery Services attachment facility (RRSAF) enables programs to communicate with DB2. Before you invoke RRSF in your program, ensure that your program satisfies any requirements for using RRSF.

When you write programs that use RRSF, ensure that they meet the following requirements:

- The program accounts for the size of the RRSF code. The RRSF code requires about 10 KB of virtual storage per address space and an additional 10 KB for each TCB that uses RRSF.
- If your local environment intercepts and replaces the z/OS LOAD SVC that RRSF uses, you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard z/OS LOAD macro. RRSF uses z/OS SVC LOAD to load a module as part of the initialization after your first service request. The module is loaded into fetch-protected storage that has the job-step protection key.

You can prepare application programs to run in RRSF similar to how you prepare applications to run in other environments, such as CICS, IMS, and TSO. You can prepare an RRSF application either in the batch environment or by using the DB2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST.

Related tasks:

Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915

How RRSF modifies the content of registers

If you do not specify the return code and reason code parameters in your RRSF function calls or if you invoke RRSF implicitly, RRSF puts a return code in register 15 and a reason code in register 0. RRSF preserves the contents of registers 2 through 14.

If you specify the return code and reason code parameters, RRSF places the return code in register 15 and in the return code parameter to accommodate high-level languages that support special return code processing.

The following table summarizes the register conventions for RRSF calls.

Table 21. Register conventions for RRSF calls

Register	Usage
R1	Parameter list pointer
R13	Address of caller's save area
R14	Caller's return address
R15	RRSAF entry point address

Implicit connections to RRSF

Resource Recovery Services attachment facility (RRSAF) establishes an implicit connection to DB2 under certain situations. The connection is established if the following are true: the RRSF language interface load module (DSNRLI) is

available, you do not explicitly specify the IDENTIFY function in a CALL DSNRLI statement in your program, and the application includes SQL statements or IFI calls.

An implicit connection causes RRSADF to initiate implicit IDENTIFY and CREATE THREAD requests to DB2. These requests are subject to the same DB2 return codes and reason codes as explicitly specified requests.

Implicit connections use the following defaults:

Subsystem name

The default name that is specified in the module DSNHDECP. RRSADF uses the installation default DSNHDECP, unless your own DSNHDECP module is in a library in a STEPLIB statement of the JOBLIB concatenation or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

Be certain that you know what the default name is and that it names the specific DB2 subsystem that you want to use.

Plan name

The member name of the database request module (DBRM) that DB2 produced when you precompiled the source program that contains the first SQL call.

Authorization ID

The 7-byte user ID that is associated with the address space, unless an authorized function has built an Accessor Environment Element (ACEE) for the address space. If an authorized function has built an ACEE, DB2 passes the 8-byte user ID from the ACEE.

For an implicit connection request, your application should not explicitly specify either the IDENTIFY function or the CREATE THREAD function. Your application can execute other explicit RRSADF calls after the implicit connection is made. An implicit connection does not perform any SIGNON processing. Your application can execute the SIGNON function at any point of consistency. To terminate an implicit connection, you must use the proper function calls.

For implicit connection requests, register 15 contains the return code, and register 0 contains the reason code. The return code and reason code are also in the message text for SQLCODE -981.

Related concepts:

“Summary of RRSADF behavior” on page 116

CALL DSNRLI statement parameter list

The CALL DSNRLI statement explicitly invokes RRSADF. When you include CALL DSNRLI statements in your program, you must specify all parameters that precede the return code parameter.

In CALL DSNRLI statements, you cannot omit any of parameters that come before the return code parameter by coding zeros or blanks. No defaults exist for those parameters for explicit connection requests. Defaults are provided for only implicit connections. All parameters starting with the return code parameter are optional.

When you want to use the default value for a parameter but specify subsequent parameters, code the CALL DSNRLI statement as follows:

- For C-language, when you code CALL DSNRLI statements in C, you need to specify the address of every parameter, using the "address of" operator (&), and not the parameter itself. For example, to pass the pklistptr parameter on the "CREATE THREAD" specify the address of the 4-byte pointer to the structure (&pklistptr):

```
fnret=dsnrli(&crthrdfn[0], &plan[0], &collid[0], &reuse[0],
            &retcode, &reascode, &pklistptr);
```

- For all languages except assembler language, code zero for that parameter in the CALL DSNRLI statement. For example, suppose that you are coding an IDENTIFY call in a COBOL program, and you want to specify all parameters except the return code parameter. You can write a statement similar to the following statement:

```
CALL 'DSNRLI' USING IDFYFN SSM RIBPTR EIBPTR TERMECB STARTECB
            BY CONTENT ZERO BY REFERENCE REASCODE.
```

- For assembler language, code a comma for that parameter in the CALL DSNRLI statement. For example, suppose that you are coding an IDENTIFY call, and you want to specify all parameters except the return code parameter. You can write a statement similar to the following statement:

```
CALL DSNRLI,(IDFYFN,SSM,RIBPTR,EIBPTR,TERMECB,STARTECB,,REASCODE)
```

For assembler programs that invoke RRSF, use a standard parameter list for an z/OS CALL. Register 1 must contain the address of a list of pointers to the parameters. Each pointer is a 4-byte address. The last address must contain the value 1 in the high-order bit.

Summary of RRSF behavior

The effect of any Resource Recovery Services attachment facility (RRS) function depends in part on what functions the program has already run. You should plan the RRS function calls that your program makes to avoid any errors and major structural problems in your application.

The following tables summarize RRS behavior after various inputs from application programs. The contents of each table cell indicate the result of calling the function in the first column for that row followed by the function in the current column heading. For example, if you issue TERMINATE THREAD and then IDENTIFY, RRS returns reason code X'00C12201'. Use these tables to understand the order in which your application must issue RRS calls, SQL statements, and IFI requests.

The RRS FIND_DB2_SYSTEMS function is omitted from these tables, because it does not affect the operation of any of the other functions

The following table summarizes RRS behavior when the next call is to the IDENTIFY function, the SWITCH TO function, the SIGNON function, or the CREATE THREAD function.

Table 22. Effect of call order when next call is IDENTIFY, SWITCH TO, SIGNON, or CREATE THREAD

Previous function	Next function			
	IDENTIFY	SWITCH TO	SIGNON, AUTH SIGNON, or CONTEXT SIGNON	CREATE THREAD
Empty: first call	IDENTIFY	X'00C12205 ¹	X'00C12204 ¹	X'00C12204 ¹
IDENTIFY	X'00F30049 ¹	Switch to <i>ssnm</i>	Signon ²	X'00C12217 ¹
SWITCH TO	IDENTIFY	Switch to <i>ssnm</i>	Signon ²	CREATE THREAD

Table 22. Effect of call order when next call is IDENTIFY, SWITCH TO, SIGNON, or CREATE THREAD (continued)

Previous function	Next function			
	IDENTIFY	SWITCH TO	SIGNON, AUTH SIGNON, or CONTEXT SIGNON	CREATE THREAD
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	X'00F30049 ¹	Switch to <i>ssnm</i>	Signon ²	CREATE THREAD
CREATE THREAD	X'00F30049 ¹	Switch to <i>ssnm</i>	Signon ²	X'00C12202 ¹
TERMINATE THREAD	X'00C12201 ¹	Switch to <i>ssnm</i>	Signon ²	CREATE THREAD
IFI	X'00F30049 ¹	Switch to <i>ssnm</i>	Signon ²	X'00C12202 ¹
SQL	X'00F30049 ¹	Switch to <i>ssnm</i>	X'00F30092 ¹³	X'00C12202 ¹
SRRCMIT or SRRBACK	X'00F30049 ¹	Switch to <i>ssnm</i>	Signon ²	X'00C12202 ¹

Notes:

1. Errors are identified by the DB2 reason code that RRSAF returns.
2. Signon means either the SIGNON function, the AUTH SIGNON function, or the CONTEXT SIGNON function.
3. The SIGNON, AUTH SIGNON, or CONTEXT SIGNON functions are not allowed if any SQL operations are requested after the CREATE THREAD function or after the last SRRCMIT or SRRBACK request.

The following table summarizes RRSAF behavior when the next call is an SQL statement or an IFI call or to the TERMINATE THREAD function, the TERMINATE IDENTIFY function, or the TRANSLATE function.

Table 23. Effect of call order when next call is SQL or IFI, TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE

Previous function	Next function			
	SQL or IFI	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
Empty: first call	SQL or IFI call ⁴	X'00C12204 ¹	X'00C12204 ¹	X'00C12204 ¹
IDENTIFY	SQL or IFI call ⁴	X'00C12203 ¹	TERMINATE IDENTIFY	TRANSLATE
SWITCH TO	SQL or IFI call ⁴	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	SQL or IFI call ⁴	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
CREATE THREAD	SQL or IFI call ⁴	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
TERMINATE THREAD	SQL or IFI call ⁴	X'00C12203 ¹	TERMINATE IDENTIFY	TRANSLATE
IFI	SQL or IFI call ⁴	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SQL	SQL or IFI call ⁴	X'00F30093 ¹²	X'00F30093 ¹³	TRANSLATE
SRRCMIT or SRRBACK	SQL or IFI call ⁴	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE

Notes:

1. Errors are identified by the DB2 reason code that RRSAF returns.
2. TERMINATE THREAD is not allowed if any SQL operations are requested after the CREATE THREAD function or after the last SRRCMIT or SRRBACK request.
3. TERMINATE IDENTIFY is not allowed if any SQL operations are requested after the CREATE THREAD function or after the last SRRCMIT or SRRBACK request.
4. If you are using an implicit connection to RRSAF and issue SQL or IFI calls, RRSAF issues implicit IDENTIFY and CREATE THREAD requests. If you continue with explicit RRSAF statements, you must follow the standard order of explicit RRSAF calls. Implicitly connecting to RRSAF does not cause an implicit SIGNON request. Therefore, you might need to issue an explicit SIGNON request to satisfy the standard order requirement. For example, an SQL statement followed by an explicit TERMINATE THREAD request results in an error. You must issue an explicit SIGNON request before issuing the TERMINATE THREAD request.

Related concepts:

➡ X'C1.....' codes (DB2 Codes)

➡ X'F3.....' codes (DB2 Codes)

RRSAF connection functions

An Resource Recovery Services attachment facility (RRSAF) connection function specifies the action that you want RRSaf to take. You specify these functions when you invoke RRSaf through CALL DSNRLI statements.

Related concepts:

“CALL DSNRLI statement parameter list” on page 115

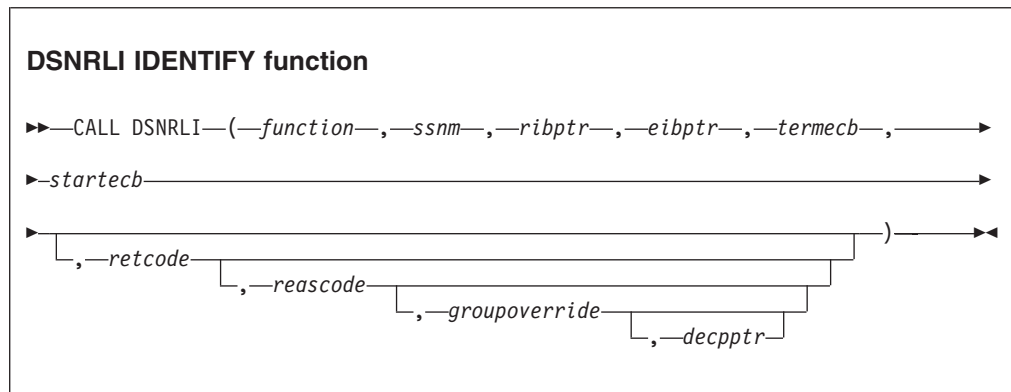
“Summary of RRSaf behavior” on page 116

IDENTIFY function for RRSaf

The RRSaf IDENTIFY function initializes a connection to DB2.

The IDENTIFY function establishes the caller's task as a user of DB2 services. If no other task in the address space currently is connected to the specified subsystem, the IDENTIFY function also initializes the address space to communicate with the DB2 address spaces. The IDENTIFY function establishes the cross-memory authorization of the address space to DB2 and builds address space control blocks.

The following diagram shows the syntax for the IDENTIFY function.



Parameters point to the following areas:

function

An 18-byte area that contains IDENTIFY followed by 10 blanks.

ssnm

A 4-byte DB2 subsystem name, or group attachment or subgroup attachment name (if used in a data sharing group) to which the connection is made. If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

ribptr

A 4-byte area in which RRSaf places the address of the release information block (RIB) after the call. You can use the RIB to determine the release level of the DB2 subsystem to which the application is connected. You can determine the modification level within the release level by examining the RIBCNUMB

and RIBCINFO fields. If the value in the RIBCNUMB field is greater than zero, check the RIBCINFO field for modification levels.

If the RIB is not available (for example, if *ssnm* names a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *ribptr* points is below the 16-MB line.

This parameter is required. However, the application does not need to refer to the returned information.

eibptr

A 4-byte area in which RRSAP places the address of the environment information block (EIB) after the call. The EIB contains environment information, such as the data sharing group, the name of the DB2 member to which the IDENTIFY request was issued, and whether the subsystem is in new-function mode. If the DB2 subsystem is not in a data sharing group, RRSAP sets the data sharing group and member names to blanks. If the EIB is not available (for example, if *ssnm* names a subsystem that does not exist), RRSAP sets the 4-byte area to zeros.

The area to which *eibptr* points is above the 16-MB line.

This parameter is required. However, the application does not need to refer to the returned information.

termecb

The address of the application's event control block (ECB) that is used for DB2 termination. DB2 posts this ECB when the system operator enters the STOP DB2 command or when DB2 is terminating abnormally. Specify a value of 0 if you do not want to use a termination ECB.

The ECB is ignored when DB2 is already stopped. The application program must examine any nonzero RRSAP or DB2 reason codes before issuing a WAIT request on this ECB.

RRSAP puts a POST code in the ECB to indicate the type of termination as shown in the following table.

Table 24. Post codes for types of DB2 termination

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

startecb

The address of the application's startup ECB. If DB2 has not started when the application issues the IDENTIFY call, DB2 posts the ECB when DB2 has started. Enter a value of zero if you do not want to use a startup ECB. DB2 posts no more than one startup ECB per address space. The ECB that is posted is associated with the most recent IDENTIFY call from that address space. The application program must examine any nonzero RRSAP or DB2 reason codes before issuing a WAIT request on this ECB.

retcode

A 4-byte area in which RRSAP places the return code.

This parameter is optional. If you do not specify *retcode*, RRSAP places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places a reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode* or its default. You can specify a default for *retcode* by specifying a comma or zero, depending on the language.

groupoverride

An 8-byte area that the application provides. This parameter is optional. If you do not want group attach to be attempted, specify 'NOGROUP'. This string indicates that the subsystem name that is specified by *ssnm* is to be used as a DB2 subsystem name, even if *ssnm* matches a group attachment or subgroup attachment name. If *groupoverride* is not provided, *ssnm* is used as the group attachment or subgroup attachment name if it matches a group attachment or subgroup attachment name.

If you specify this parameter in any language except assembler, you must also specify the *retcode* and *reascode* parameters. In assembler language, you can omit the *retcode* and *reascode* parameters by specifying commas as place-holders.

Recommendation: Avoid using the *groupoverride* parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex. However, you should use this parameter in a data sharing environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.

decpptr

A 4-byte area in which RRSF is to put the address of the DSNHDECP or a user-specified application defaults module that was loaded by subsystem *ssnm* when that subsystem was started. This 4-byte area is a 31-bit pointer. If *ssnm* is not found, the 4-byte area is set to 0.

The area to which *decpptr* points is above the 16-MB line.

If you specify this parameter in any language except assembler, you must also specify the *retcode*, *reascode*, and *groupoverride* parameters. In assembler language, you can omit the *retcode*, *reascode*, and *groupoverride* parameters by specifying commas as placeholders.

Example of RRSF IDENTIFY function calls

The following table shows an IDENTIFY call in each language.

Table 25. Examples of RRSF IDENTIFY calls

Language	Call example
Assembler	CALL DSNRLI, (IDFYFN,SSNM,RIBPTR,EIBPTR,TERM ECB,STARTECB, RETCODE,REASCODE,GRPOVER,DECPTR)
C ¹	fnret=dsnrli(&idfyfn[0],&ssnm[0], &ribptr, &eibptr, &term ECB, &start ECB, &retcode, &reascode,&grpover[0],&decpptr);
COBOL	CALL 'DSNRLI' USING IDFYFN SSNM RIBPTR EIBPTR TERM ECB STARTECB RETCODE REASCODE GRPOVER DECPTR.
Fortran	CALL DSNRLI (IDFYFN,SSNM,RIBPTR,EIBPTR,TERM ECB,STARTECB, RETCODE,REASCODE,GRPOVER,DECPTR)
PL/I ¹	CALL DSNRLI (IDFYFN,SSNM,RIBPTR,EIBPTR,TERM ECB,STARTECB, RETCODE,REASCODE,GRPOVER,DECPTR);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAP.

Internal processing for the IDENTIFY function

When you call the IDENTIFY function, DB2 performs the following steps:

1. DB2 determines whether the user address space is authorized to connect to DB2. DB2 invokes the z/OS SAF and passes a primary authorization ID to SAF. That authorization ID is the 7-byte user ID that is associated with the address space, unless an authorized function has built an ACEE for the address space. If an authorized function has built an ACEE, DB2 passes the 8-byte user ID from the ACEE. SAF calls an external security product, such as RACF, to determine if the task is authorized to use the following items:
 - The DB2 resource class (CLASS=DSNR)
 - The DB2 subsystem (SUBSYS=*ssnm*)
 - Connection type RRSAP
2. If that check is successful, DB2 calls the DB2 connection exit routine to perform additional verification and possibly change the authorization ID.
3. DB2 searches for a matching trusted context in the system cache and then the catalog based on the following criteria:
 - The primary authorization ID matches a trusted context SYSTEM AUTHID.
 - The job or started task name matches the JOBNAME attribute that is defined for the identified trusted context.

If a trusted context is defined, DB2 checks if SECURITY LABEL is defined in the trusted context. If SECURITY LABEL is defined, DB2 verifies the SECURITY LABEL with RACF by using the RACROUTE VERIFY request. This security label is used to verify multi-level security for SYSTEM AUTHID. If a matching trusted context is defined, DB2 establishes the connection as trusted. Otherwise, the connection is established without any additional privileges.

4. DB2 then sets the connection name to RRSAP and the connection type to RRSAP.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

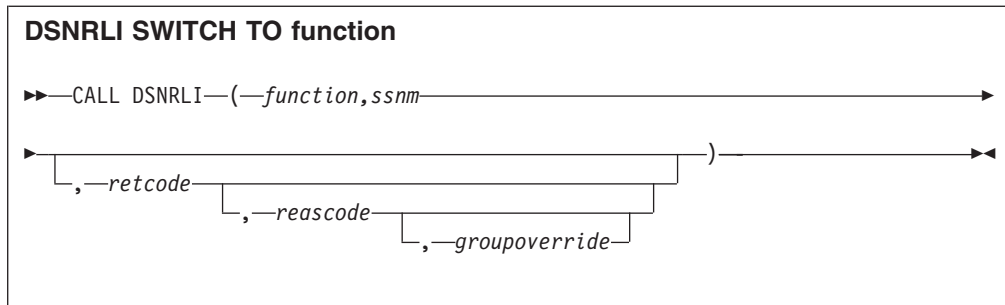
SWITCH TO function for RRSAP

The RRSAP SWITCH TO function directs RRSAP, SQL, or IFI requests to a specified DB2 subsystem. Use the SWITCH TO function to establish connections to multiple DB2 subsystems from a single task.

The SWITCH TO function is useful only after a successful IDENTIFY call. If you have established a connection with one DB2 subsystem, you must issue a SWITCH TO call before you make an IDENTIFY call to another DB2 subsystem. Otherwise, DB2 returns return code X'200' and reason code X'00C12201'.

The first time that you make a SWITCH TO call to a new DB2 subsystem, DB2 returns return code 4 and reason code X'00C12205' as a warning to indicate that the current task has not yet been identified to the new DB2 subsystem.

The following diagram shows the syntax for the SWITCH TO function.



Parameters point to the following areas:

function

An 18-byte area that contains SWITCH TO followed by nine blanks.

ssnm

| A 4-byte DB2 subsystem name, or group attachment or subgroup attachment
 | name (if used in a data sharing group) to which the connection is made. If
 | *ssnm* is less than four characters long, pad it on the right with blanks to a
 | length of four characters.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

groupoverride

| An 8-byte area that the application provides. This parameter is optional. If you
 | do not want group attach to be attempted, specify 'NOGROUP'. This string
 | indicates that the subsystem name that is specified by *ssnm* is to be used as a
 | DB2 subsystem name, even if *ssnm* matches a group attachment or subgroup
 | attachment name. If *groupoverride* is not provided, *ssnm* is used as the group
 | attachment or subgroup attachment name if it matches a group attachment or
 | subgroup attachment name.

| If you specify this parameter in any language except assembler, you must also
 | specify the *retcode* and *reascode* parameters. In assembler language, you can
 | omit the *retcode* and *reascode* parameters by specifying commas as
 | place-holders.

| **Recommendation:** Avoid using the *groupoverride* parameter when possible,
 | because it limits the ability to do dynamic workload routing in a Parallel
 | Sysplex. However, you should use this parameter in a data sharing
 | environment when you want to connect to a specific member of a data sharing
 | group, and the subsystem name of that member is the same as the group
 | attachment or subgroup attachment name.

Examples

Examples of RRSAF SWITCH TO calls: The following table shows a SWITCH TO call in each language.

Table 26. Examples of RRSAF SWITCH TO calls

Language	Call example
Assembler	CALL DSNRLI,(SWITCHFN,SSNM,RETCODE,REASCODE,GRPOVER)
C ¹	fnret=dsnrli(&switchfn[0], &ssnm[0], &retcode, &reascode,&grpover[0]);
COBOL	CALL 'DSNRLI' USING SWITCHFN RETCODE REASCODE GRPOVER.
Fortran	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER)
PL/I ¹	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER);

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Example of using the SWITCH TO function to interact with multiple DB2 subsystems: The following example shows how you can use the SWITCH TO function to interact with three DB2 subsystems.

```
RRSAF calls for subsystem db21:
  IDENTIFY
  SIGNON
  CREATE THREAD
Execute SQL on subsystem db21
SWITCH TO db22
IF retcode = 4 AND reascode = '00C12205'X THEN
  DO;
    RRSAF calls on subsystem db22:
    IDENTIFY
    SIGNON
    CREATE THREAD
  END;
Execute SQL on subsystem db22
SWITCH TO db23
IF retcode = 4 AND reascode = '00C12205'X THEN
  DO;
    RRSAF calls on subsystem db23:
    IDENTIFY
    SIGNON
    CREATE THREAD
  END;
Execute SQL on subsystem 23
SWITCH TO db21
Execute SQL on subsystem 21
SWITCH TO db22
Execute SQL on subsystem 22
SWITCH TO db21
Execute SQL on subsystem 21
SRRRCMIT (to commit the UR)
SWITCH TO db23
Execute SQL on subsystem 23
SWITCH TO db22
Execute SQL on subsystem 22
SWITCH TO db21
Execute SQL on subsystem 21
SRRRCMIT (to commit the UR)
```

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

SIGNON function for RRSAF

The RRSAF SIGNON function establishes a primary authorization ID and, optionally, one or more secondary authorization IDs for a connection.

Requirement: Your program does not need to be an authorized program to issue the SIGNON call. For that reason, before you issue the SIGNON call, you must issue the RACF external security interface macro RACROUTE REQUEST=VERIFY to perform the following actions:

- Define and populate an ACEE to identify the user of the program.
- Associate the ACEE with the user's TCB.
- Verify that the user is defined to RACF and authorized to use the application.

Generally, you issue a SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a SIGNON call if the application is at a point of consistency, and one of the following conditions is true:

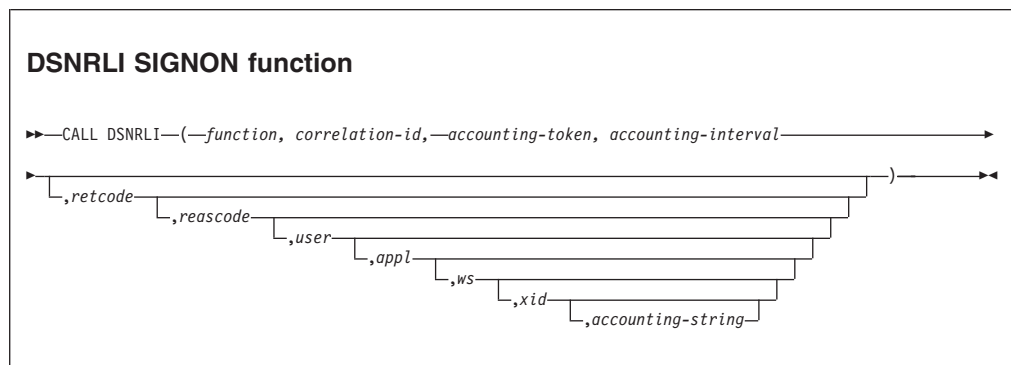
- The value of *reuse* in the CREATE THREAD call was RESET.
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEP DYNAMIC(NO), and all special registers are at their initial state. If open held cursors exist or the package or plan is bound with KEEP DYNAMIC(YES), you can issue a SIGNON call only if the primary authorization ID has not changed.

After you issue a SIGNON call, subsequent SQL statements return an error (SQLCODE -900) if the both of following conditions are true:

- The connection was established as trusted when it was initialized.
- The primary authorization ID that was used when you issued the SIGNON call is not allowed to use the trusted connection.

If a trusted context is defined, DB2 checks if SECURITY LABEL is defined in the trusted context. If SECURITY LABEL is defined, DB2 verifies the security label with RACF by using the RACROUTE VERIFY request. This security label is used to verify multi-level security for SYSTEM AUTHID.

The following diagram shows the syntax for the SIGNON function.



Parameters point to the following areas:

function

An 18-byte area that contains SIGNON followed by twelve blanks.

correlation-id

A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is

displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in the output from the DISPLAY THREAD command. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token

A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. Setting the value of the accounting token sets the value of the CURRENT CLIENT_ACCTNG special register. If *accounting-token* is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

Alternatively, you change the value of the DB2 accounting token with RRSAP functions AUTH SIGNON, CONTEXT SIGNON or SET_CLIENT_ID. You can retrieve the DB2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

accounting-interval

A 6-byte area that specifies when DB2 writes an accounting record.

If you specify COMMIT in that area, DB2 writes an accounting record each time that the application issues SRRRCMIT without open held cursors. If the accounting interval is COMMIT and an SRRRCMIT is issued while a held cursor is open, the accounting interval spans that commit and ends at the next valid accounting interval end point (such as the next SRRRCMIT that is issued without open held cursors, application termination, or SIGNON with a new authorization ID).

If you specify any other value, DB2 writes an accounting record when the application terminates or when you call the SIGNON function with a new authorization ID.

retcode

A 4-byte area in which RRSAP places the return code.

This parameter is optional. If you do not specify *retcode*, RRSAP places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSAP places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSAP places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

user

A 16-byte area that contains the user ID of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays this user ID in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the user ID sets the value of the CURRENT CLIENT_USERID special register. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This parameter is optional. If you specify *user*, you must also specify *retcode* and *reascode*. If you do not specify *user*, no user ID is associated with the connection.

appl

A 32-byte area that contains the application or transaction name of the user's application. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays the application name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the application name sets the value of the CURRENT CLIENT_APPLNAME special register. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This parameter is optional. If you specify *appl*, you must also specify *retcode*, *reascode*, and *user*. If you do not specify *appl*, no application or transaction is associated with the connection.

ws An 18-byte area that contains the workstation name of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays the workstation name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the workstation name sets the value of the CURRENT CLIENT_WRKSTNNAME special register. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This field is optional. If you specify *ws*, you must also specify *retcode*, *reascode*, *user*, and *appl*. If you do not specify *ws*, no workstation name is associated with the connection.

xid

A 4-byte area that indicates whether the thread is part of a global transaction. A DB2 thread that is part of a global transaction can share locks with other DB2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

You can specify one of the following values for *xid*:

- 0** Indicates that the thread is not part of a global transaction. The value 0 must be specified as a binary integer.
- 1** Indicates that the thread is part of a global transaction and that DB2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID. The value 1 must be specified as a binary integer. Alternatively, if you want DB2 to return the generated global transaction ID to the caller, specify an address instead of 1.

address The 4-byte address of an area in which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify.

Alternatively, if you want DB2 to generate and return a global transaction ID, pass the address of a null global transaction ID by setting the *format ID* field of the global transaction ID to binary -1 ('FFFFFF'X). DB2 then replaces the contents of the area with the generated transaction ID. The area at the specified address must be in writable storage and have a length of at least 140 bytes to accommodate the largest possible transaction ID value.

The following table shows the format of a global transaction ID.

Table 27. Format of a user-created global transaction ID

Field description	Length in bytes	Data type
Format ID	4	Integer
Global transaction ID length (1 - 64)	4	Integer
Branch qualifier length (1 - 64)	4	Integer
Global transaction ID	1 to 64	Character
Branch qualifier	0 to 64	Character

accounting-string

A one-byte length field and a 255-byte area in which you can put a value for a DB2 accounting string. This value is placed in the DDF accounting trace records in the QMDASQLI field, which is mapped by DSNDQMDA DSECT. If *accounting-string* is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

This parameter is optional. If you specify *accounting-string*, you must also specify *retcode*, *reascode*, *user*, *appl* and *xid*. If you do not specify *accounting-string*, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of *accounting-string* can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUFIX field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

Example of RRSF SIGNON calls

The following table shows a SIGNON call in each language.

Table 28. Examples of RRSF SIGNON calls

Language	Call example
assembler	CALL DSNRLI,(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME,XIDPTR)
C ¹	fnret=dsnrli(&sgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0], &xidptr);
COBOL	CALL 'DSNRLI' USING SGNONFN CORRID ACCTTKN ACCTINT RETCODE REASCODE USERID APPLNAME WSNAME XIDPTR.
Fortran	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME,XIDPTR)
PL/I ¹	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME,XIDPTR);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSF.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

Related reference:

➡ RACROUTE REQUEST=VERIFY (standard form) (Security Server RACROUTE Macro Reference)

AUTH SIGNON function for RRSF

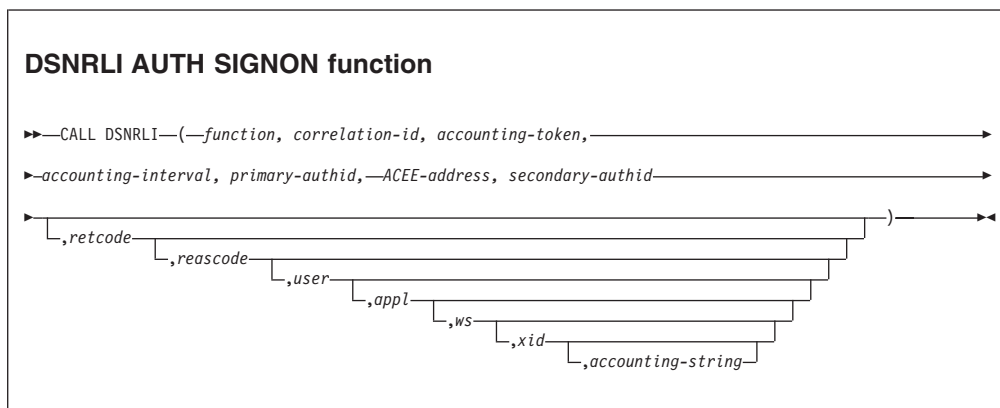
The RRSF AUTH SIGNON function enables an APF authorization program to pass an ID to DB2.

An APF-authorized program can pass to DB2 either a primary authorization ID and, optionally, one or more secondary authorization IDs, or an ACEE that is used for authorization checking. These IDs are then associated with the connection.

Generally, you issue an AUTH SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue an AUTH SIGNON call if the application is at a point of consistency, and one of the following conditions is true:

- The value of *reuse* in the CREATE THREAD call was RESET.
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEP DYNAMIC(NO), and all special registers are at their initial state. If open held cursors exist or the package or plan is bound with KEEP DYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

The following diagram shows the syntax for the AUTH SIGNON function.



Parameters point to the following areas:

function

An 18-byte area that contains AUTH SIGNON followed by seven blanks.

correlation-id

A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the DISPLAY THREAD command. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token

A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. Setting the

value of the accounting token sets the value of the CURRENT CLIENT_ACCTNG special register. If *accounting-token* is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

You can also change the value of the DB2 accounting token with RRSF functions SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID. You can retrieve the DB2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

accounting-interval

A 6-byte area with that specifies when DB2 writes an accounting record.

If you specify COMMIT in that area, DB2 writes an accounting record each time that the application issues SRRRCMIT without open held cursors. If the accounting interval is COMMIT and an SRRRCMIT is issued while a held cursor is open, the accounting interval spans that commit and ends at the next valid accounting interval end point (such as the next SRRRCMIT that is issued without open held cursors, application termination, or SIGNON with a new authorization ID).

If you specify any other value, DB2 writes an accounting record when the application terminates or when you call the SIGNON function with a new authorization ID.

primary-authid

An 8-byte area in which you can put a primary authorization ID. If you are not passing the authorization ID to DB2 explicitly, put X'00' or a blank in the first byte of the area.

ACEE-address

The 4-byte address of an ACEE that you pass to DB2. If you do not want to provide an ACEE, specify 0 in this field.

secondary-authid

An 8-byte area in which you can put a secondary authorization ID. If you do not pass the authorization ID to DB2 explicitly, put X'00' or a blank in the first byte of the area. If you enter a secondary authorization ID, you must also enter a primary authorization ID.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

user

A 16-byte area that contains the user ID of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays this user ID in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the user ID sets the value of the CURRENT CLIENT_USERID

special register. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This parameter is optional. If you specify *user*, you must also specify *retcode* and *reascde*. If you do not specify this parameter, no user ID is associated with the connection.

appl

A 32-byte area that contains the application or transaction name of the user's application. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays the application name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the application name sets the value of the CURRENT_CLIENT_APPLNAME special register. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This parameter is optional. If you specify *appl*, you must also specify *retcode*, *reascde*, and *user*. If you do not specify this parameter, no application or transaction is associated with the connection.

ws An 18-byte area that contains the workstation name of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays the workstation name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the workstation name sets the value of the CURRENT_CLIENT_WRKSTNNAME special register. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This parameter is optional. If you specify *ws*, you must also specify *retcode*, *reascde*, *user*, and *appl*. If you do not specify this parameter, no workstation name is associated with the connection.

You can also change the value of the workstation name with RRSAF functions SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID. You can retrieve the workstation name with the CURRENT_CLIENT_WRKSTNNAME special register.

xid

A 4-byte area that indicates whether the thread is part of a global transaction. A DB2 thread that is part of a global transaction can share locks with other DB2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

You can specify one of the following values for *xid*:

- 0** Indicates that the thread is not part of a global transaction. The value 0 must be specified as a binary integer.
- 1** Indicates that the thread is part of a global transaction and that DB2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID. The value 1 must be specified as a binary integer. Alternatively, if you want DB2 to return the generated global transaction ID to the caller, specify an address instead of 1.

address The 4-byte address of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread

becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify.

Alternatively, if you want DB2 to generate and return a global transaction ID, pass the address of a null global transaction ID by setting the *format ID* field of the global transaction ID to binary -1 ('FFFFFF'X). DB2 then replaces the contents of the area with the generated transaction ID. The area at the specified address must be in writable storage and have a length of at least 140 bytes to accommodate the largest possible transaction ID value.

The format of a global transaction ID is shown in the description of the RRSF SIGNON function.

accounting-string

A 1-byte length field and a 255-byte area in which you can put a value for a DB2 accounting string. This value is placed in the DDF accounting trace records in the QMDASQLI field, which is mapped by DSNDQMDA DSECT. If *accounting-string* is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

This parameter is optional. If you specify this *accounting-string*, you must also specify *retcode*, *reascde*, *user*, *appl*, and *xid*. If you do not specify this parameter, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of *accounting-string* can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUFX field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

Example of RRSF AUTH SIGNON calls

The following table shows a AUTH SIGNON call in each language.

Table 29. Examples of RRSF AUTH SIGNON calls

Language	Call example
Assembler	CALL DSNRLI,(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEPTR, SAUTHID,RETCODE,REASCODE, USERID,APPLNAME,WSNAME,XIDPTR)
C ¹	fnret=dsnrli(&asgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &pauthid[0], &aceptr, &sauthid[0], &retcode, &reascde, &userid[0], &applname[0], &wsname[0], &xidptr);
COBOL	CALL 'DSNRLI' USING ASGNONFN CORRID ACCTTKN ACCTINT PAUTHID ACEPTR SAUTHID RETCODE REASCODE USERID APPLNAME WSNAME XIDPTR.
Fortran	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEPTR, SAUTHID,RETCODE,REASCODE,USERID, APPLNAME,WSNAME,XIDPTR)
PL/I ¹	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEPTR, SAUTHID,RETCODE,REASCODE,USERID, APPLNAME,WSNAME,XIDPTR);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

Related reference:

“SIGNON function for RRSAF” on page 124

CONTEXT SIGNON function for RRSAF

The RRSAF CONTEXT SIGNON function establishes a primary authorization ID and one or more secondary authorization IDs for a connection.

Requirement: Before you invoke CONTEXT SIGNON, you must have called the RRS context services function Set Context Data (CTXSDTA) to store a primary authorization ID and optionally, the address of an ACEE in the context data whose context key you supply as input to CONTEXT SIGNON.

The CONTEXT SIGNON function uses the context key to retrieve the primary authorization ID from data that is associated with the current RRS context. DB2 uses the RRS context services function Retrieve Context Data (CTXRDTA) to retrieve context data that contains the authorization ID and ACEE address. The context data must have the following format:

Version number

A 4-byte area that contains the version number of the context data. Set this area to 1.

Server product name

An 8-byte area that contains the name of the server product that set the context data.

ALET A 4-byte area that can contain an ALET value. DB2 does not reference this area.

ACEE address

A 4-byte area that contains an ACEE address or 0 if an ACEE is not provided. DB2 requires that the ACEE is in the home address space of the task.

If you pass an ACEE address, the CONTEXT SIGNON function uses the value in ACEEGRP as the secondary authorization ID if the length of the group name (ACEEGRP) is not 0.

primary-authid

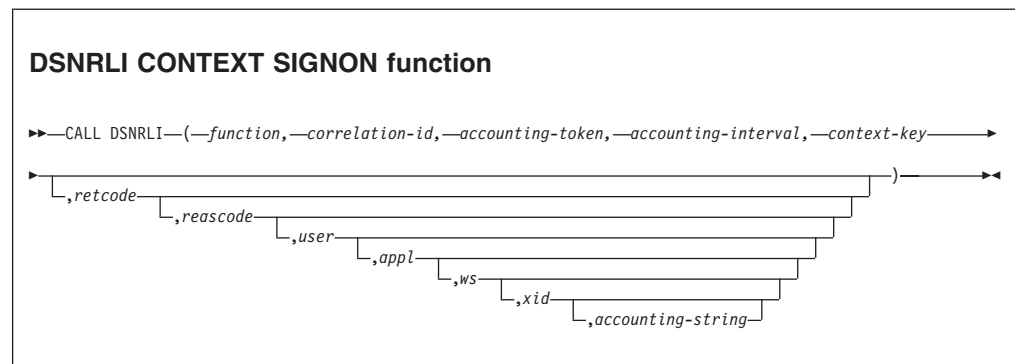
An 8-byte area that contains the primary authorization ID to be used. If the authorization ID is less than 8 bytes in length, pad it on the right with blank characters to a length of 8 bytes.

If the new primary authorization ID is not different than the current primary authorization ID (which was established when the IDENTIFY function was invoked or at a previous SIGNON invocation), DB2 invokes only the signon exit. If the value has changed, DB2 establishes a new primary authorization ID and new SQL authorization ID and then invokes the signon exit.

Generally, you issue a CONTEXT SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a CONTEXT SIGNON call if the application is at a point of consistency, and one of the following conditions is true:

- The value of *reuse* in the CREATE THREAD call was RESET.
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEP DYNAMIC(NO), and all special registers are at their initial state. If open held cursors exist or the package or plan is bound with KEEP DYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

The following diagram shows the syntax for the CONTEXT SIGNON function.



Parameters point to the following areas:

function

An 18-byte area that contains CONTEXT SIGNON followed by four blanks.

correlation-id

A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the DISPLAY THREAD command. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token

A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. Setting the value of the accounting token sets the value of the CURRENT CLIENT_ACCTNG special register. If *accounting-token* is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

You can also change the value of the DB2 accounting token with RRSF functions SIGNON, AUTH SIGNON, or SET_CLIENT_ID. You can retrieve the DB2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

accounting-interval

A 6-byte area that specifies when DB2 writes an accounting record.

If you specify COMMIT in that area, DB2 writes an accounting record each time that the application issues SRRCMIT without open held cursors. If the accounting interval is COMMIT and an SRRCMIT is issued while a held cursor is open, the accounting interval spans that commit and ends at the next valid

accounting interval end point (such as the next SRRRCMIT that is issued without open held cursors, application termination, or SIGNON with a new authorization ID).

If you specify any other value, DB2 writes an accounting record when the application terminates or when you call the SIGNON function with a new authorization ID.

context-key

A 32-byte area in which you put the context key that you specified when you called the RRS Set Context Data (CTXSDTA) service to save the primary authorization ID and an optional ACEE address.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

user

A 16-byte area that contains the user ID of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays this user ID in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the user ID sets the value of the CURRENT_CLIENT_USERID special register. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This parameter is optional. If you specify *user*, you must also specify *retcode* and *reascode*. If you do not specify *user*, no user ID is associated with the connection.

appl

A 32-byte area that contains the application or transaction name of the user's application. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays the application name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the application name sets the value of the CURRENT_CLIENT_APPLNAME special register. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This parameter is optional. If you specify *appl*, you must also specify *retcode*, *reascode*, and *user*. If you do not specify *appl*, no application or transaction is associated with the connection.

ws

An 18-byte area that contains the workstation name of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. DB2 displays the workstation name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. Setting the workstation name sets the value of the CURRENT_CLIENT_WRKSTNNAME special register. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This parameter is optional. If you specify *ws*, you must also specify *retcode*, *reascode*, *user*, and *appl*. If you do not specify *ws*, no workstation name is associated with the connection.

You can also change the value of the workstation name with the RRSF functions SIGNON, AUTH SIGNON, or SET_CLIENT_ID. You can retrieve the workstation name with the CLIENT_WRKSTNNAME special register.

xid

A 4-byte area that indicates whether the thread is part of a global transaction. A DB2 thread that is part of a global transaction can share locks with other DB2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

You can specify one of the following values for *xid*:

- 0 Indicates that the thread is not part of a global transaction. The value 0 must be specified as a binary integer.
- 1 Indicates that the thread is part of a global transaction and that DB2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID. The value 1 must be specified as a binary integer. Alternatively, if you want DB2 to return the generated global transaction ID to the caller, specify an address instead of 1.

address The 4-byte address of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify.

Alternatively, if you want DB2 to generate and return a global transaction ID, pass the address of a null global transaction ID by setting the *format ID* field of the global transaction ID to binary -1 ('FFFFFF'X). DB2 then replaces the contents of the area with the generated transaction ID. The area at the specified address must be in writable storage and have a length of at least 140 bytes to accommodate the largest possible transaction ID value.

The format of a global transaction ID is shown in the description of the RRSF SIGNON function.

accounting-string

A one-byte length field and a 255-byte area in which you can put a value for a DB2 accounting string. This value is placed in the DDF accounting trace records in the QMDASQLI field, which is mapped by DSNDQMDA DSECT. If *accounting-string* is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

This parameter is optional. If you specify this *accounting-string*, you must also specify *retcode*, *reascode*, *user*, *appl* and *xid*. If you do not specify this parameter, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of *accounting-string* can contain a maximum of 200 characters. The QMDASFLN

field contains the accounting suffix length, and the QMDASUFEX field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

Example of RRSF CONTEXT SIGNON calls

The following table shows a CONTEXT SIGNON call in each language.

Table 30. Examples of RRSF CONTEXT SIGNON calls

Language	Call example
Assembler	CALL DSNRLI, (CSGNONFN, CORRID, ACCTTKN, ACCTINT, CTXTKEY, RETCODE, REASCODE, USERID, APPLNAME, WNAME, XIDPTR)
C ¹	fnret=dsnrli(&csgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &ctxtkey[0], &retcode, &reascodes, &userid[0], &applname[0], &wname[0], &xidptr);
COBOL	CALL 'DSNRLI' USING CSGNONFN CORRID ACCTTKN ACCTINT CTXTKEY RETCODE REASCODE USERID APPLNAME WNAME XIDPTR.
Fortran	CALL DSNRLI (CSGNONFN, CORRID, ACCTTKN, ACCTINT, CTXTKEY, RETCODE, REASCODE, USERID, APPLNAME, WNAME, XIDPTR)
PL/I ¹	CALL DSNRLI (CSGNONFN, CORRID, ACCTTKN, ACCTINT, CTXTKEY, RETCODE, REASCODE, USERID, APPLNAME, WNAME, XIDPTR);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSF.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

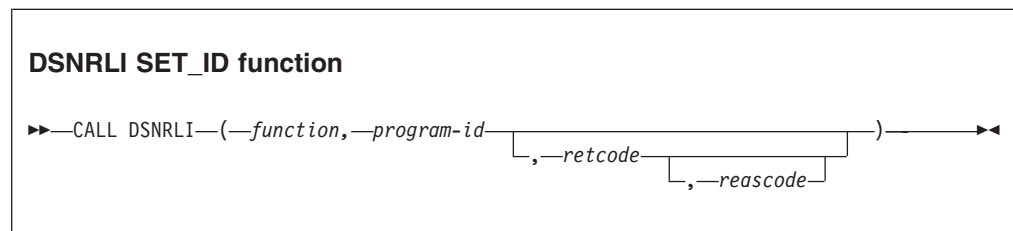
Related reference:

“SIGNON function for RRSF” on page 124

SET_ID function for RRSF

The RRSF SET_ID function sets a new value for the client program ID that can be used to identify the user. The function then passes this information to DB2 when the next SQL request is processed.

The following diagram shows the syntax of the SET_ID function.



Parameters point to the following areas:

function

An 18-byte area that contains SET_ID followed by 12 blanks.

program-id

An 80-byte area that contains the caller-provided string to be passed to DB2. If *program-id* is less than 80 characters, you must pad it with blanks on the right to a length of 80 characters.

DB2 places the contents of *program-id* into IFCID 316 records, along with other statistics, so that you can identify which program is associated with a particular SQL statement.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode* RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

Example of RRSF SET_ID calls

The following table shows a SET_ID call in each language.

Table 31. Examples of RRSF SET_ID calls

Language	Call example
Assembler	CALL DSNRLI, (SETIDFN, PROGID, RETCODE, REASCODE)
C ¹	fnret=dsnrli(&setidfn[0], &progid[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING SETIDFN PROGID RETCODE REASCODE.
Fortran	CALL DSNRLI (SETIDFN, PROGID, RETCODE, REASCODE)
PL/I ¹	CALL DSNRLI (SETIDFN, PROGID, RETCODE, REASCODE);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSF.

Related tasks:

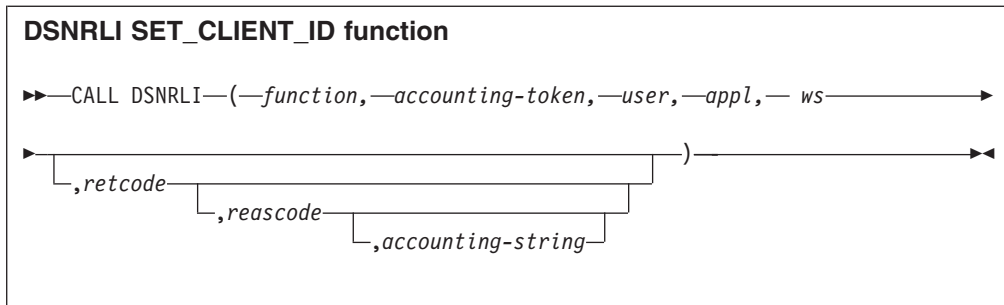
“Invoking the Resource Recovery Services attachment facility” on page 106

SET_CLIENT_ID function for RRSF

The RRSF SET_CLIENT_ID function sets new values for the client user ID, the application program name, the workstation name, the accounting token, and the DDF client accounting string. The function then passes this information to DB2 when the next SQL request is processed.

These values can be used to identify the end user. The calling program defines the contents of these parameters. DB2 places the parameter values in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records.

The following diagram shows the syntax of the SET_CLIENT_ID function.



Parameters point to the following areas:

function

An 18-byte area that contains SET_CLIENT_ID followed by 5 blanks.

accounting-token

A 22-byte area in which you can put a value for a DB2 accounting token. This value is placed in the DB2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. If *accounting-token* is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

Alternatively, you can change the value of the DB2 accounting token with the RRSF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the DB2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

user

A 16-byte area that contains the user ID of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 places this user ID in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

You can also change the value of the client user ID with the RRSF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the client user ID with the CLIENT_USERID special register.

appl

An 32-byte area that contains the application or transaction name of the end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 places the application name in the output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. If *appl* is less than 32 characters, you must pad it on the right with blanks to a length of 32 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

You can also change the value of the application name with the RRSF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the application name with the CLIENT_APPLNAME special register.

ws An 18-byte area that contains the workstation name of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 places this workstation name in the

output from the DISPLAY THREAD command and in DB2 accounting and statistics trace records. If *ws* is less than 18 characters, you must pad it on the right with blanks to a length of 18 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

You can also change the value of the workstation name with the RRSAF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the workstation name with the CLIENT_WRKSTNNAME special register.

retcode

A 4-byte area in which RRSAF places the return code.

You can omit this parameter by specifying a value of 0 in the parameter list.

This parameter is optional. If you do not specify *retcode*, RRSAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSAF places the reason code.

You can omit this parameter by specifying a value of 0 in the parameter list.

This parameter is optional. If you do not specify *reascode*, RRSAF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

accounting-string

A one-byte length field and a 255-byte area in which you can put a value for a DB2 accounting string. This value is placed in the DDF accounting trace records in the QMDASUFx field, which is mapped by DSNDQMDA DSECT. If *accounting-string* is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

You can omit this parameter by specifying a value of 0 in the parameter list.

This parameter is optional. If you specify this *accounting-string*, you must also specify *retcode*, *reascode*, *user*, and *appl*. If you do not specify this parameter, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSAF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of *accounting-string* can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUFx field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

Example of RRSAF SET_CLIENT_ID calls

The following table shows a SET_CLIENT_ID call in each language.

Table 32. Examples of RRSAF SET_CLIENT_ID calls

Language	Call example
Assembler	CALL DSNRLI, (SECLIDFN, ACCT, USER, APPL, WS, RETCODE, REASCODE)
C ¹	fnret=dsnrli(&seclidfn[0], &acct[0], &user[0], &appl[0], &ws[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING SECLIDFN ACCT USER APPL WS RETCODE REASCODE.

Table 32. Examples of RRSF SET_CLIENT_ID calls (continued)

Language	Call example
Fortran	CALL DSNRLI (SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE)
PL/I ¹	CALL DSNRLI (SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSF.

Related tasks:

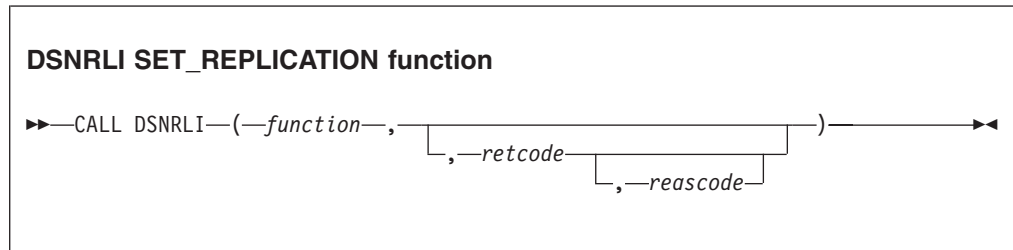
“Invoking the Resource Recovery Services attachment facility” on page 106

SET_REPLICATION function for RRSF

The RRSF SET_REPLICATION function enables an APF authorized program to identify to DB2 as a replication program.

Calling the SET_REPLICATION function is optional. If you do not call it, DB2 treats the application normally. The SET_REPLICATION function allows the application to perform insert, update, and delete operations then the tablespace or database is started access RREPL.

The following diagram shows the syntax for the SET REPLICATION function.



Parameters point to the following areas:

function

An 18-byte area that contains SET_REPLICATION.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places a reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

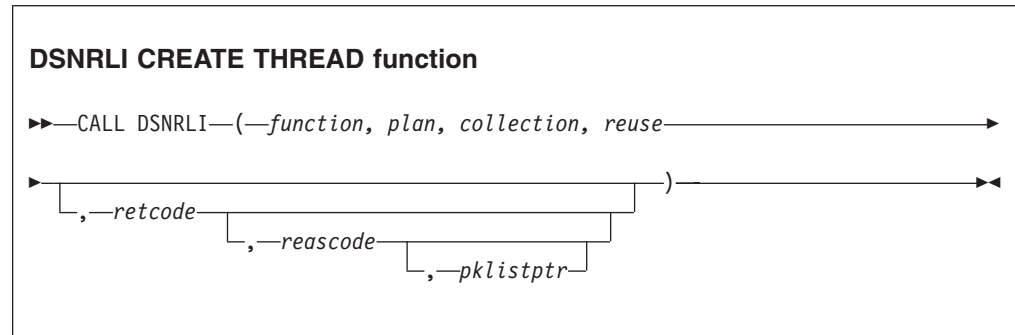
Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

CREATE THREAD function for RRSAF

The RRSAF CREATE THREAD function allocates the DB2 resources that are required for an application to issue SQL or IFI requests. This function must complete before the application can execute SQL statements or IFI requests.

The following diagram shows the syntax of the CREATE THREAD function.



Parameters point to the following areas:

function

An 18-byte area that contains CREATE THREAD followed by five blanks.

plan

An 8-byte DB2 plan name. RRSAF allocates the named plan.

If you provide a collection name instead of a plan name, specify the question mark character (?) in the first byte of this field. DB2 then allocates a special plan named ?RRSAF and uses the value that you specify for *collection*. When DB2 allocates a plan named ?RRSAF, DB2 checks authorization to execute the package in the same way as it checks authorization to execute a package from a requester other than DB2 for z/OS.

If you do not provide a collection name in the *collection* field, you must enter a valid plan name in this field.

collection

An 18-byte area in which you enter a collection name. DB2 uses the collection names to locate a package that is associated with the first SQL statement in the program.

When you provide a collection name and put the question mark character (?) in the *plan* field, DB2 allocates a plan named ?RRSAF and a package list that contains the following two entries:

- The specified collection name.
- An entry that contains * for the location, collection name, and package name. (This entry lets the application access remote locations and access packages in collections other than the default collection that is specified at create thread time.)

The application can use the SET CURRENT PACKAGESET statement to change the collection ID that DB2 uses to locate a package.

If you provide a plan name in the *plan* field, DB2 ignores the value in the *collection* field.

reuse

An 8-byte area that controls the action that DB2 takes if a SIGNON call is issued after a CREATE THREAD call. Specify one of the following values in this field:

RESET

Releases any held cursors and reinitializes the special registers

INITIAL

Does not allow the SIGNON call

This parameter is required. If the 8-byte area does not contain either RESET or INITIAL, the default value is INITIAL.

retcode

A 4-byte area in which RRSAP places the return code.

This parameter is optional. If you do not specify *retcode*, RRSAP places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSAP places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSAP places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

pklistptr

A 4-byte field that contains a pointer to a user-supplied data area that contains a list of collection IDs. A collection ID is an SQL identifier of 1 to 128 letters, digits, or the underscore character that identifies a collection of packages. The length of the data area is a maximum of 2050 bytes. The data area contains a 2-byte length field, followed by up to 2048 bytes of collection ID entries, separated by commas.

When you specify *pklistptr* and the question mark character (?) in the *plan* field, DB2 allocates a special plan named ?RRSAP and a package list that contains the following entries:

- The collection names that you specify in the data area to which *pklistptr* points
- An entry that contains * for the location, collection ID, and package name

If you also specify *collection*, DB2 ignores that value.

Each collection entry must be of the form *collection-ID.**, **.collection-ID.**, or **.*.*.collection-ID* and must follow the naming conventions for a collection ID, as described in the description of the BIND and REBIND options.

DB2 uses the collection names to locate a package that is associated with the first SQL statement in the program. The entry that contains **.*.** lets the application access remote locations and access packages in collections other than the default collection that is specified at create thread time.

The application can use the SET CURRENT PACKAGESET statement to change the collection ID that DB2 uses to locate a package.

This parameter is optional. If you specify this parameter, you must also specify *retcode* and *reascode*.

If you provide a plan name in the *plan* field, DB2 ignores the *pklistptr* value.

Recommendation: Using a package list can have a negative impact on performance. For better performance, specify a short package list.

Example of RRSAF CREATE THREAD calls

The following table shows a CREATE THREAD call in each language.

Table 33. Examples of RRSAF CREATE THREAD calls

Language	Call example
Assembler	CALL DSNRLI,(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLSTPTR)
C ¹	fnret=dsnrli(&crthrdfn[0], &plan[0], &collid[0], &reuse[0], &retcode, &reascodes, &pklistptr);
COBOL	CALL 'DSNRLI' USING CRTHRDFN PLAN COLLID REUSE RETCODE REASCODE PKLSTPTR.
Fortran	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLSTPTR)
PL/I ¹	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLSTPTR);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

➡ Authorizing plan or package access through applications (Managing Security)

Related reference:

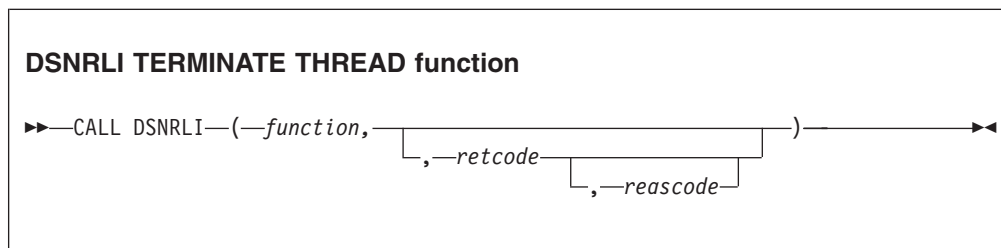
➡ BIND and REBIND options for packages and plans (DB2 Commands)

TERMINATE THREAD function for RRSAF

The RRSAF TERMINATE THREAD function deallocates DB2 resources that are associated with a plan and were previously allocated for an application by the CREATE THREAD function. You can then use the CREATE THREAD function to allocate another plan with the same connection.

If you call the TERMINATE THREAD function and the application is not at a point of consistency, RRSAF returns reason code X'00C12211'.

The following diagram shows the syntax of the TERMINATE THREAD function.



Parameters point to the following areas:

function

An 18-byte area the contains TERMINATE THREAD followed by two blanks.

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

Example of RRSF TERMINATE THREAD calls

The following table shows a TERMINATE THREAD call in each language.

Table 34. Examples of RRSF TERMINATE THREAD calls

Language	Call example
Assembler	CALL DSNRLI, (TRMTHDFN, RETCODE, REASCODE)
C ¹	fnret=dsnrli(&trmthdfn[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING TRMTHDFN RETCODE REASCODE.
Fortran	CALL DSNRLI (TRMTHDFN, RETCODE, REASCODE)
PL/I ¹	CALL DSNRLI (TRMTHDFN, RETCODE, REASCODE);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSF.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

TERMINATE IDENTIFY function for RRSF

The RRSF TERMINATE IDENTIFY function terminates a connection to DB2. Calling the TERMINATE IDENTIFY function is optional. If you do not call it, DB2 performs the same functions when the task terminates.

If DB2 terminates, the application must issue TERMINATE IDENTIFY to reset the RRSF control blocks. This action ensures that future connection requests from the task are successful when DB2 restarts.

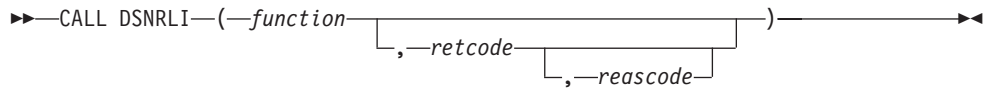
The TERMINATE IDENTIFY function removes the calling task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures that were created for the address space and removes the cross-memory authorization.

If the application is not at a point of consistency when you call the TERMINATE IDENTIFY function, RRSF returns reason code X'00C12211'.

If the application allocated a plan, and you call the TERMINATE IDENTIFY function without first calling the TERMINATE THREAD function, DB2 deallocates the plan before terminating the connection.

The following diagram shows the syntax of the TERMINATE IDENTIFY function.

DSNRLI TERMINATE IDENTIFY function



Parameters point to the following areas:

function

An 18-byte area that contains TERMINATE IDENTIFY.

retcode

A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSAF places the return code in register 15 and the reason code in register 0.

reascde

A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify *reascde*, RRSAF places the reason code in register 0.

If you specify *reascde*, you must also specify *retcode*.

Example of RRSAF TERMINATE IDENTIFY calls

The following table shows a TERMINATE IDENTIFY call in each language.

Table 35. Examples of RRSAF TERMINATE IDENTIFY calls

Language	Call example
Assembler	CALL DSNRLI, (TMIDFYFN, RETCODE, REASCODE)
C ¹	fnret=dsnrli(&tmidfyfn[0], &retcode, &reascde);
COBOL	CALL 'DSNRLI' USING TMIDFYFN RETCODE REASCODE.
Fortran	CALL DSNRLI(TMIDFYFN, RETCODE, REASCODE)
PL/I ¹	CALL DSNRLI(TMIDFYFN, RETCODE, REASCODE);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

TRANSLATE function for RRSAF

The RRSAF TRANSLATE function converts a hexadecimal reason code for a DB2 error into a signed integer SQL code and a printable error message. The SQL code and message text are placed in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

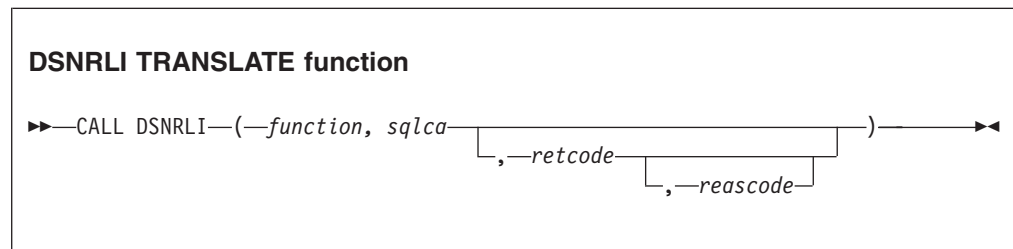
Consider the following rules and recommendations about when to use and not use the TRANSLATE function:

- You cannot call the TRANSLATE function from the Fortran language.
- Call the TRANSLATE function only after a successful IDENTIFY operation. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.
- The TRANSLATE function translates codes that begin with X'00F3', but it does not translate RRSF reason codes that begin with X'00C1'.

If you receive error reason code X'00F30040' (resource unavailable) after an OPEN request, the TRANSLATE function returns the name of the unavailable database object in the last 44 characters of the SQLERRM field.

If the TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original DB2 function code and the return and error reason codes in the SQLERRM field. The contents of registers 0 and 15 do not change, unless TRANSLATE fails. In this case, register 0 is set to X'00C12204', and register 15 is set to 200.

The following diagram shows the syntax of the TRANSLATE function.



Parameters point to the following areas:

function

An 18-byte area that contains the word TRANSLATE followed by nine blanks.

sqlca

The program's SQL communication area (SQLCA).

retcode

A 4-byte area in which RRSF places the return code.

This parameter is optional. If you do not specify *retcode*, RRSF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSF places the reason code.

This parameter is optional. If you do not specify *reascode*, RRSF places the reason code in register 0.

If you specify *reascode*, you must also specify *retcode*.

Example of RRSAF TRANSLATE calls

The following table shows a TRANSLATE call in each language.

Table 36. Examples of RRSAF TRANSLATE calls

Language	Call example
Assembler	CALL DSNRLI, (XLATFN, SQLCA, RETCODE, REASCODE)
C ¹	fnret=dsnrli(&connfn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING XLATFN SQLCA RETCODE REASCODE.
PL/I ¹	CALL DSNRLI (XLATFN, SQLCA, RETCODE, REASCODE);

Note:

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

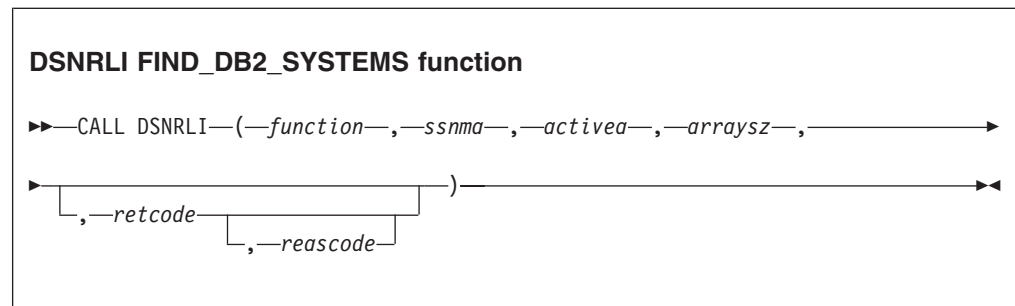
Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

FIND_DB2_SYSTEMS function for RRSAF

The RRSAF FIND_DB2_SYSTEMS function identifies all active DB2 subsystems on a z/OS LPAR.

The following diagram shows the syntax of the FIND_DB2_SYSTEMS function.



Parameters point to the following areas:

function

An 18-byte area that contains FIND_DB2_SYSTEMS followed by two blanks.

ssnma

A storage area for an array of 4-byte character strings into which RRSAF places the names of all the DB2 subsystems (SSIDs) that are defined for the current LPAR. You must provide the storage area. If the array is larger than the number of DB2 subsystems, RRSAF returns the value ' ' (four blanks) in all unused array members.

activea

A storage area for an array of 4-byte values into which RRSAF returns an indication of whether a defined subsystem is active. Each value is represented as a fixed 31-bit integer. The value 1 means that the subsystem is active. The value 0 means that the subsystem is not active. The size of this array must be

the same as the size of the *ssnma* array. If the array is larger than the number of DB2 subsystems, RRSAF returns the value -1 in all unused array members.

The information in the *activea* array is the information that is available at the point in time that you requested it and might change at any time.

arraysz

A 4-byte area, represented as a fixed 31-bit integer, that specifies the number of entries for the *ssnma* and *activea* arrays. If the number of array entries is insufficient to contain all of the subsystems defined on the current LPAR, RRSAF uses all available entries and returns return code 4.

retcode

A 4-byte area in which RRSAF is to place the return code for this call to the FIND_DB2_SYSTEMS function.

This parameter is optional. If you do not *retcode*, RRSAF places the return code in register 15 and the reason code in register 0.

reascode

A 4-byte area in which RRSAF is to place the reason code for this call to the FIND_DB2_SYSTEMS function.

This parameter is optional. If you do not specify *reascode*, RRSAF places the reason code in register 0.

Example values that the FIND_DB2_SYSTEMS function returns

Assume that two subsystems are defined on the current LPAR. Subsystem DB2A is active, and subsystem DB2B is stopped. Suppose that you invoke RRSAF with the function FIND_DB2_SYSTEMS and a value of 3 for *arraysz*. The *ssnma* array and *activea* array are set to the following values:

Table 37. Example values returned in the *ssnma* and *activea* arrays

Array element number	Values in <i>ssnma</i> array	Values in <i>activea</i> array
1	DB2A	1
2	DB2B	0
3	(four blanks)	-1

Related tasks:

“Invoking the Resource Recovery Services attachment facility” on page 106

RRSAF return codes and reason codes

If you specify return code and reason code parameters in an Resource Recovery Services attachment facility (RRSAF) function call, RRSAF returns the return code and reason code in those parameters. If you do not specify those parameters or implicitly invoke RRSAF, RRSAF puts the return code in register 15 and the reason code in register 0.

When the reason code begins with X'00F3', except for X'00F30006', you can use the RRSAF TRANSLATE function to obtain error message text that can be printed and displayed.

For SQL calls, RRSAF returns standard SQL return codes in the SQLCA. RRSAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

The following table lists the RRSAF return codes.

Table 38. RRSAF return codes

Return code	Explanation
0	The call completed successfully.
4	Status information is available. See the reason code for details.
>4	The call failed. See the reason code for details.

Related reference:

“TRANSLATE function for RRSAF” on page 145

Sample RRSAF scenarios

One or more tasks can use Resource Recovery Services attachment facility (RRSAF) to connect to DB2. This connection can be made either implicitly or explicitly. For explicit connections, a task calls one or more of the RRSAF connection functions.

A single task

The following example pseudocode illustrates a single task running in an address space that explicitly connects to DB2 through RRSAF. z/OS RRS controls commit processing when the task terminates normally.

```
IDENTIFY
SIGNON
CREATE THREAD
SQL or IFI
:
TERMINATE IDENTIFY
```

Multiple tasks

In the following scenario, multiple tasks in an address space explicitly connect to DB2 through RRSAF. Task 1 executes no SQL statements and makes no IFI calls. Its purpose is to monitor DB2 termination and startup ECBs and to check the DB2 release level.

TASK 1	TASK 2	TASK 3	TASK n
IDENTIFY	IDENTIFY	IDENTIFY	IDENTIFY
	SIGNON	SIGNON	SIGNON
	CREATE THREAD	CREATE THREAD	CREATE THREAD
	SQL	SQL	SQL

	SRRCMIT	SRRCMIT	SRRCMIT
	SQL	SQL	SQL

	SRRCMIT	SRRCMIT	SRRCMIT

TERMINATE IDENTIFY			

Reusing a DB2 thread

The following example pseudocode shows a DB2 thread that is reused by another user at a point of consistency. When the application calls the SIGNON function for user B, DB2 reuses the plan that is allocated by the CREATE THREAD function for user A.

```

IDENTIFY
SIGNON user A
CREATE THREAD
  SQL
  ...
  SRRCMIT
SIGNON user B
  SQL
  ...
  SRRCMIT

```

Switching DB2 threads between tasks

The following scenario shows how you can switch the threads for four users (A, B, C, and D) among two tasks (1 and 2).

Task 1	Task 2
<pre> CTXBEGC (create context a) CTXSWCH(a,0) IDENTIFY SIGNON user A CREATE THREAD (Plan A) SQL ... CTXSWCH(0,a) </pre>	<pre> CTXBEGC (create context b) CTXSWCH(b,0) IDENTIFY SIGNON user B CREATE THREAD (plan B) SQL ... CTXSWCH(0,b) </pre>
<pre> CTXBEGC (create context c) CTXSWCH(c,0) IDENTIFY SIGNON user C CREATE THREAD (plan C) SQL ... CTXSWCH(b,c) SQL (plan B) ... </pre>	<pre> CTXBEGC (create context d) CTXSWCH(d,0) IDENTIFY SIGNON user D CREATE THREAD (plan D) SQL ... CTXSWCH(0,d) ... CTXSWCH(a,0) SQL (plan A) </pre>

The applications perform the following steps:

- Task 1 creates context a, switches contexts so that context a is active for task 1, and calls the IDENTIFY function to initialize a connection to a subsystem. A task must always call the IDENTIFY function before a context switch can occur. After the IDENTIFY operation is complete, task 1 allocates a thread for user A, and performs SQL operations.

At the same time, task 2 creates context b, switches contexts so that context b is active for task 2, calls the IDENTIFY function to initialize a connection to the subsystem, allocates a thread for user B, and performs SQL operations.

When the SQL operations complete, both tasks perform RRS context switch operations. Those operations disconnect each DB2 thread from the task under which it was running.

- Task 1 then creates context c, calls the IDENTIFY function to initialize a connection to the subsystem, switches contexts so that context c is active for task 1, allocates a thread for user C, and performs SQL operations for user C.

Task 2 does the same operations for user D.

- When the SQL operations for user C complete, task 1 performs a context switch operation to perform the following actions:
 - Switch the thread for user C away from task 1.
 - Switch the thread for user B to task 1.

For a context switch operation to associate a task with a DB2 thread, the DB2 thread must have previously performed an IDENTIFY operation. Therefore, before the thread for user B can be associated with task 1, task 1 must have performed an IDENTIFY operation.

- Task 2 performs two context switch operations to perform the following actions:
 - Disassociate the thread for user D from task 2.
 - Associate the thread for user A with task 2.

Program examples for RRSAF

The Resource Recovery Services attachment facility (RRSAF) enables programs to communicate with DB2. You can use RRSAF as an alternative to CAF.

Example JCL for invoking RRSAF

The following sample JCL shows how to use RRSAF in a batch environment. The DSNRRSAF DD statement starts the RRSAF trace. Use that DD statement only if you are diagnosing a problem.

```
//jobname      JOB      z/OS_jobcard_information
//RRSJCL       EXEC     PGM=RRS_application_program
//STEPLIB      DD      DSN=application_load_library
//             DD      DSN=DB2_load_library

:

//SYSPRINT     DD      SYSOUT=*
//DSNRRSAF     DD      DUMMY
//SYSUDUMP     DD      SYSOUT=*
```

Example of loading and deleting the RRSAF language interface

The following code segment shows how an application loads entry points DSNRLI and DSNHLIR of the RRSAF language interface. Storing the entry points in variables LIRLI and LISQL ensures that the application loads the entry points only once. Delete the loaded modules when the application no longer needs to access DB2.

```
***** GET LANGUAGE INTERFACE ENTRY ADDRESSES
      LOAD EP=DSNRLI          Load the RRSAF service request EP
      ST   R0,LIRLI          Save this for RRSAF service requests
      LOAD EP=DSNHLIR        Load the RRSAF SQL call Entry Point
      ST   R0,LISQL          Save this for SQL calls
*
*   .   Insert connection service requests and SQL calls here
*   .
      DELETE EP=DSNRLI        Correctly maintain use count
      DELETE EP=DSNHLIR      Correctly maintain use count
```

Example of using dummy entry point DSNHLI for RRSAF

Each of the DB2 attachment facilities contains an entry point named DSNHLI. When you use RRSAF but do not specify the ATTACH(RRSAF) precompiler option, the precompiler generates BALR instructions to DSNHLI for SQL statements in your program. To find the correct DSNHLI entry point without including DSNRLI in your load module, code a subroutine, with entry point DSNHLI, that passes control to entry point DSNHLIR in the DSNRLI module. DSNHLIR is unique to DSNRLI and is at the same location as DSNHLI in DSNRLI. DSNRLI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, the intermediate subroutine must account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
*****
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
*****
      DS      0D
DSNHLI CSECT          Begin CSECT
      STM    R14,R12,12(R13)  Prologue
      LA     R15,SAVEHLI      Get save area address
      ST     R13,4(,R15)      Chain the save areas
      ST     R15,8(,R13)      Chain the save areas
      LR     R13,R15          Put save area address in R13
      L      R15,LISQL        Get the address of real DSNHLI
      BASSM R14,R15          Branch to DSNRLI to do an SQL call
*                               DSNRLI is in 31-bit mode, so use
*                               BASSM to assure that the addressing
*                               mode is preserved.
      L      R13,4(,R13)      Restore R13 (caller's save area addr)
      L      R14,12(,R13)     Restore R14 (return address)
      RETURN (1,12)          Restore R1-12, NOT R0 and R15 (codes)
```

Example of connecting to DB2 with RRSF

This example uses the variables that are declared in the following code.

```
***** VARIABLES SET BY APPLICATION *****
LIRLI  DS    F           DSNRLI entry point address
LISQL  DS    F           DSNHLIR entry point address
SSNM   DS    CL4        DB2 subsystem name for IDENTIFY
CORRID DS    CL12       Correlation ID for SIGNON
ACCTTKN DS CL22        Accounting token for SIGNON
ACCTINT DS    CL6       Accounting interval for SIGNON
PLAN   DS    CL8        DB2 plan name for CREATE THREAD
COLLID DS    CL18       Collection ID for CREATE THREAD.  If
*                               PLAN contains a plan name, not used.
REUSE  DS    CL8        Controls SIGNON after CREATE THREAD
CONTROL DS CL8          Action that application takes based
*                               on return code from RRSF
***** VARIABLES SET BY DB2 *****
STARTECB DS    F           DB2 startup ECB
TERMECB  DS    F           DB2 termination ECB
EIBPTR   DS    F           Address of environment info block
RIBPTR   DS    F           Address of release info block
***** CONSTANTS *****
CONTINUE DC    CL8'CONTINUE' CONTROL value: Everything OK
IDFYFN   DC    CL18'IDENTIFY'  ' Name of RRSF service
SGNONFN  DC    CL18'SIGNON'    ' Name of RRSF service
CRTHDFN  DC    CL18'CREATE THREAD' ' Name of RRSF service
TRMTHDFN DC    CL18'TERMINATE THREAD' ' Name of RRSF service
TMIDFYFN DC    CL18'TERMINATE IDENTIFY' Name of RRSF service
***** SQLCA and RIB *****
      EXEC SQL INCLUDE SQLCA
              DSNDRIB          Map the DB2 Release Information Block
***** Parameter list for RRSF calls *****
RRSAFCLL CALL  ,(*,*,*,*,*,*,*),VL,MF=L
```

The following example code shows how to issue requests for the RRSF functions IDENTIFY, SIGNON, CREATE THREAD, TERMINATE THREAD, and TERMINATE IDENTIFY. This example does not show a task that waits on the DB2 termination ECB. You can code such a task and use the z/OS WAIT macro to monitor the ECB. The task that waits on the termination ECB should detach the sample code if the termination ECB is posted. That task can also wait on the DB2 startup ECB. This example waits on the startup ECB at its own task level.


```

***** IDENTIFY *****
      L    R15,LIRLI          Get the Language Interface address
      CALL (15),(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB),VL,MF=X
      (E,RRSAFCLL)
      BAL  R14,CHEKCODE      Call a routine (not shown) to check
*                               return and reason codes
      CLC  CONTROL,CONTINUE  Is everything still OK
      BNE  EXIT              If CONTROL not 'CONTINUE', stop loop
      USING R8,RIB          Prepare to access the RIB
      L    R8,RIBPTR        Access RIB to get DB2 release level
      CLC  RIBREL,RIBR999   DB2 V10 or later?
      BE   USERELX         If RIBREL = '999', use RIBRELX
      WRITE 'The current DB2 release level is' RIBREL
      B    SIGNON          Continue with signon
      USERELX WRITE 'The current DB2 release level is' RIBRELX
***** SIGNON *****
      SIGNON L    R15,LIRLI          Get the Language Interface address
      CALL  (15),(SGNONFN,CORRID,ACCTKN,ACCTINT),VL,MF=(E,RRSAFCLL)
      BAL   R14,CHEKCODE          Check the return and reason codes
***** CREATE THREAD *****
      L    R15,LIRLI          Get the Language Interface address
      CALL (15),(CRTHDFN,PLAN,COLLID,REUSE),VL,MF=(E,RRSAFCLL)
      BAL  R14,CHEKCODE      Check the return and reason codes
***** SQL *****
*                               Insert your SQL calls here.  The DB2 Precompiler
*                               generates calls to entry point DSNHLI.  You should
*                               code a dummy entry point of that name to intercept
*                               all SQL calls.  A dummy DSNHLI is shown in the following
*                               section.
***** TERMINATE THREAD *****
      CLC  CONTROL,CONTINUE  Is everything still OK?
      BNE  EXIT              If CONTROL not 'CONTINUE', shut down
      L    R15,LIRLI          Get the Language Interface address
      CALL (15),(TRMTHDFN),VL,MF=(E,RRSAFCLL)
      BAL  R14,CHEKCODE      Check the return and reason codes
***** TERMINATE IDENTIFY *****
      CLC  CONTROL,CONTINUE  Is everything still OK
      BNE  EXIT              If CONTROL not 'CONTINUE', stop loop
      L    R15,LIRLI          Get the Language Interface address
      CALL (15),(TMIDFYFN),VL,MF=(E,RRSAFCLL)
      BAL  R14,CHEKCODE      Check the return and reason codes

```

Universal language interface

The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

The following figure shows the general structure of DSNULI and a program that uses it:

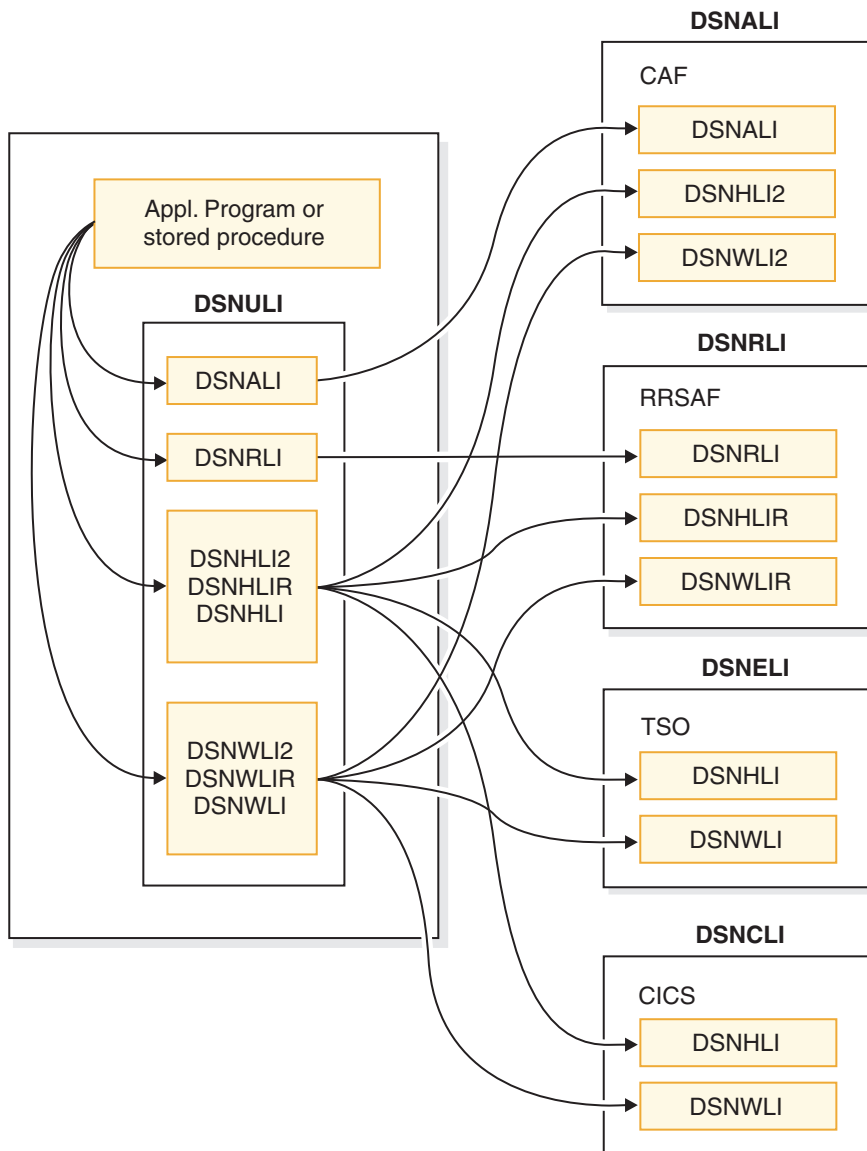


Figure 4. Application program or stored procedure linked with DSNULI

DSNULI has no aliases. The module has nine entry points: DSNALI, DSNRLI, DSNCLI, DSNHLI, DSNHLI2, DSNHLIR, DSNWLI, DSNWLI2, and DSNWLIR. DSNULI will dynamically load and branch to the appropriate language interface module, based on the entry point name (for attachment-specific entry points), or based on the current environment (for the generic entry points DSNHLI and DSNWLI).

- Entry point DSNALI handles explicit DB2 Call Attach Facility connection service requests.
- Entry point DSNRLI handles explicit DB2 Resource Recovery Services Attach Facility connection service requests.
- Entry point DSNCLI is provided for link-editing with CICS
- DSNHLI, DSNHLI2, and DSNHLIR handle SQL calls. DSNHLI2 is an explicit SQL call via the Call Attachment Facility. DSNHLIR is an explicit SQL call via the Resource Recovery Services Attachment Facility. They are provided for compatibility only. Applications designed to run in any environment should use the generic entry point, DSNHLI.

- DSNWLI, DSNWLI2, DSNWLIR handle IFI calls. DSNWLI2 is an explicit IFI call via the Call Attachment Facility. DSNWLIR is an explicit IFI call via the Resource Recovery Services Attachment Facility. They are provided for compatibility only. Applications designed to run in any environment should use the generic entry point, DSNWLI.

Link-editing an application with DSNULI

To create a single load module that can be used in more than one attachment environment, you can link-edit your program or stored procedure with the Universal Language Interface module (DSNULI) instead of with one of the environment-specific language interface modules (DSNELI, DSNALI, DSNRLI, or DSNCLI).

DSNULI should be link-edited with TSO, CAF, RRSF applications (including Stored Procedures), and CICS applications. DSNULI does not support dynamic loading or IMS applications. DSNULI determines the run time environment, then dynamically loads and branches to the appropriate language interface module (DSNELI, DSNALI, DSNRLI, or DSNCLI).

Considerations:

- If maximum performance is the primary requirement, link-edit with DSNELI, DSNALI, DSNRLI, or DSNCLI rather than DSNULI. If maintaining a single copy of a load module is the primary requirement, link-edit with DSNULI.
- If CAF implicit connect functionality is required, link-edit your application with DSNALI instead of with DSNULI. DSNULI defaults to RRSF implicit connections if an attachment environment has not been established upon entry to DSNHLL. Attachment environments are established by calling DSNRLI or DSNALI initially, or by running an SQL application under the TSO command processor or under CICS.
- DSNULI will not explicitly delete the loaded DSNELI, DSNALI, DSNRLI or DSNCLI. If an application cannot tolerate having these modules deleted only at task termination, use DSNELI, DSNALI, DSNRLI or DSNCLI instead of DSNULI.
- DSNULI is shipped with the linkage attributes AMODE(31) and RMODE(ANY) and must be entered in AMODE(31).

To link-edit an application with DSNULI:

You can include DSNULI when you link-edit your load module. For example, you can use a linkage editor control statement like this in your JCL:

```
INCLUDE SYSLIB(DSNULI)
```

By coding this statement, you avoid linking to one of the environment-specific language interface modules.

Controlling the CICS attachment facility from an application

Use the CICS attachment facility to access DB2 from CICS application programs.

You can start and stop the CICS attachment facility from within an application program.

To control the CICS attachment facility:

1. To start the CICS attachment facility, perform one of the following actions:

- Include the following statement in your application:
EXEC CICS LINK PROGRAM('DSN2COM0')
 - Use the system programming interface SET DB2CONN for the CICS Transaction Server.
2. To stop the CICS attachment facility, perform one of the following actions:
- Include the following statement in your application:
EXEC CICS LINK PROGRAM('DSN2COM2')
 - Use the system programming interface SET DB2CONN for the CICS Transaction Server.

Related information:

 SET DB2CONN (CICS Transaction Server for z/OS)

Detecting whether the CICS attachment facility is operational

Before you execute SQL statements in a CICS program, you should determine if the CICS attachment facility is available. You do not need to do this test if the CICS attachment facility is started and you are using standby mode.

When an SQL statement is executed, and the CICS attachment facility is in standby mode, the attachment issues SQLCODE -923 with a reason code that indicates that DB2 is not available.

To detect whether the CICS attachment facility is operational:

Use the INQUIRE EXITPROGRAM command for the CICS Transaction Server in your application.

The following example shows how to use this command. In this example, the INQUIRE EXITPROGRAM command tests whether the resource manager for SQL, DSNCSQL, is up and running. CICS returns the results in the EIBRESP field of the EXEC interface block (EIB) and in the field whose name is the argument of the CONNECTST parameter (in this case, STST). If the EIBRESP value indicates that the command completed normally and the STST value indicates that the resource manager is available, you can then execute SQL statements.

```

STST      DS      F
ENTNAME   DS      CL8
EXITPROG  DS      CL8
:
:
          MVC     ENTNAME,=CL8'DSNCSQL'
          MVC     EXITPROG,=CL8'DSN2EXT1'
EXEC CICS INQUIRE EXITPROGRAM(EXITPROG)
          ENTRYNAME(ENTNAME) CONNECTST(STST) NOHANDLE
          CLC     EIBRESP,DFHRESP(NORMAL)
          BNE     NOTREADY
          CLC     STST,DFHVALUE(CONNECTED)
          BNE     NOTREADY
UPNREADY  DS      0H
          attach is up
NOTREADY  DS      0H
          attach is not up yet

```

If you use the INQUIRE EXITPROGRAM command to avoid AEY9 abends and the CICS attachment facility is down, the storm drain effect can occur. The *storm drain effect* is a condition that occurs when a system continues to receive work, even though that system is down.

Related concepts:

 Storm-drain effect (DB2 Installation and Migration)

Related information:

 [INQUIRE EXITPROGRAM \(CICS Transaction Server for z/OS\)](#)

 [-923 \(DB2 Codes\)](#)

Improving thread reuse in CICS applications

Having transactions reuse threads is generally recommended because each thread creation is associated with a high processor cost.

To improve thread reuse in CICS applications:

Close all cursors that are declared with the WITH HOLD option before each sync point. DB2 does not automatically close them. A thread for an application that contains an open cursor cannot be reused. You should close all cursors immediately after you finish using them.

Related concepts:

“Held and non-held cursors” on page 725

Chapter 3. Coding SQL statements in application programs: General information

A *query* is an SQL statement that returns data from a DB2 database. Your program can communicate this SQL statement to DB2 in one of several ways. After processing the statement, DB2 issues a return code, which your program should then test to determine the result of the operation.

To include DB2 queries in an application program:

1. Choose one of the following methods for communicating with DB2:
 - Static SQL
 - Embedded dynamic SQL
 - Open Database Connectivity (ODBC)
 - JDBC application support
 - SQLJ application support

ODBC lets you access data through ODBC function calls in your application. You execute SQL statements by passing them to DB2 through a ODBC function call. ODBC eliminates the need for precompiling and binding your application and increases the portability of your application by using the ODBC interface.

If you are writing your applications in Java, you can use JDBC application support to access DB2. JDBC is similar to ODBC but is designed specifically for use with Java. In addition to using JDBC, you can use SQLJ application support to access DB2. SQLJ is designed to simplify the coding of DB2 calls for Java applications.

2. Optional: Declare the tables and views that you use. You can use DCLGEN to generate these declarations.
3. Define the items that your program can use to check whether an SQL statement executed successfully. You can either define an SQL communications area (SQLCA) or declare SQLSTATE and SQLCODE host variables.
4. Define at least one SQL descriptor area (SQLDA).
5. Declare any of the following data items for passing data between DB2 and a host language:
 - host variables
 - host variable arrays
 - host structures




Ensure that you use the appropriate data types.

6. Code SQL statements to access DB2 data. Ensure that you delimit these statements properly.

Consider using cursors to select a set of rows and then process the set either one row at a time or one rowset at a time.

7. Check the execution of the SQL statements.
8. Handle any SQL error codes.

Related concepts:

-  Introduction to DB2 ODBC (DB2 Programming for ODBC)
-  JDBC application programming (DB2 Application Programming for Java)
-  SQLJ application programming (DB2 Application Programming for Java)

Related tasks:

“Delimiting an SQL statement” on page 183

“Including dynamic SQL in your program” on page 193

“Retrieving a set of rows by using a cursor” on page 722

 Programming applications for performance (DB2 Performance)

Declaring table and view definitions

Before your program issues SQL statements that select, insert, update, or delete data, the program needs to declare the tables and views that those statements access.

Your program is not required to declare tables or views, but doing so offers the following advantages:

- Clear documentation in the program
The declaration specifies the structure of the table or view and the data type of each column. You can refer to the declaration for the column names and data types in the table or view.
- Assurance that your program uses the correct column names and data types
The DB2 precompiler uses your declarations to make sure that you have used correct column names and data types in your SQL statements. The DB2 precompiler issues a warning message when the column names and data types in SQL statements do not correspond to the table and view declarations in your program.

To declare table and view definitions:

Perform one of the following actions:

- Include an SQL DECLARE TABLE statement in your program. Specify the name of the table or view and list each column and its data type.

When you declare a table or view that contains a column with a distinct type, declare that column with the source type of the distinct type rather than with the distinct type itself. When you declare the column with the source type, DB2 can check embedded SQL statements that reference that column at precompile time.

In a COBOL program, code the DECLARE TABLE statement in the WORKING-STORAGE SECTION or LINKAGE SECTION within the DATA DIVISION.

Example DECLARE statement in a COBOL program: The following DECLARE TABLE statement in a COBOL program defines the DSN8A10.DEPT table:

```
EXEC SQL
  DECLARE DSN8A10.DEPT TABLE
    (DEPTNO   CHAR(3)           NOT NULL,
     DEPTNAME VARCHAR(36)       NOT NULL,
     MGRNO    CHAR(6)           ,
     ADMRDEPT CHAR(3)           NOT NULL,
     LOCATION CHAR(16)         )
END-EXEC.
```

- Use DCLGEN, the declarations generator that is supplied with DB2, to create these declarations for you and then include them in your program.

Restriction: You can use DCLGEN for only C, COBOL, and PL/I programs.

Related reference:

 [DECLARE TABLE \(DB2 SQL\)](#)

DCLGEN (declarations generator)

Your program should declare the tables and views that it accesses. The DB2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

DCLGEN generates a table or view declaration and puts it into a member of a partitioned data set that you can include in your program. When you use DCLGEN to generate a table declaration, DB2 gets the relevant information from the DB2 catalog. The catalog contains information about the table or view definition and the definition of each column within the table or view. DCLGEN uses this information to produce an SQL DECLARE TABLE statement for the table or view and a corresponding PL/I or C structure declaration or COBOL record description.

Related reference:

 [DCLGEN \(DECLARATIONS GENERATOR\) \(DSN\) \(DB2 Commands\)](#)

Generating table and view declarations by using DCLGEN

Your program should declare the tables and views that it accesses. For C, COBOL, and PL/I programs, you can use DCLGEN to produce these declarations, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

Requirements:

- DB2 must be active before you can use DCLGEN.
- You can use DCLGEN for table declarations only if the table or view that you are declaring already exists.
- If you use DCLGEN, you must use it before you precompile your program.

To generate table and view declarations by using DCLGEN:

1. Invoke DCLGEN by performing one of the following actions:
 - **To start DCLGEN from ISPF through DB2I:** Select the DCLGEN option on the DB2I Primary Option Menu panel. Then follow the detailed instructions for generating table and view declarations by using DCLGEN from DB2I.
 - **To start DCLGEN directly from TSO:** Sign on to TSO, issue the TSO command DSN, and then issue the subcommand DCLGEN.
 - **To start DCLGEN directly from a CLIST:** From a CLIST, running in TSO foreground or background, issue DSN and then DCLGEN.
 - **To start DCLGEN with JCL:** Supply the required information in JCL and run DCLGEN in batch. Use the sample jobs DSNTEJ2C and DSNTEJ2P in the *prefix.SDSNSAMP* library as models.

Requirement: If you want to start DCLGEN in the foreground and your table names include DBCS characters, you must provide and display double-byte characters. If you do not have a terminal that displays DBCS characters, you can enter DBCS characters by using the hex mode of ISPF edit.

DCLGEN creates the declarations in the specified data set.

DCLGEN generates a table or column name in the DECLARE statement as a non-delimited identifier unless at least one of the following conditions is true:




- The name contains special characters and is not a DBCS string.
- The name is a DBCS string, and you have requested delimited DBCS names.

2. If you use an SQL reserved word as an identifier, edit the DCLGEN output to add the appropriate SQL delimiters.
3. Make any other necessary edits to the DCLGEN output.

DCLGEN produces output that is intended to meet the needs of most users, but occasionally, you need to edit the DCLGEN output to work in your specific case. For example, DCLGEN is unable to determine whether a column that is defined as NOT NULL also contains the DEFAULT clause, so you must edit the DCLGEN output to add the DEFAULT clause to the appropriate column definitions.

DCLGEN produces declarations based on the encoding scheme of the source table. Therefore, if your application uses a different encoding scheme, you might need to manually adjust the declarations. For example, if your source table is in EBCDIC with CHAR columns and your application is in COBOL, DCLGEN produces declarations of type PIC X. However, suppose your host variables in your COBOL application are UTF-16. In this case, you will need to manually change the declarations to be type PIC N USAGE NATIONAL.

Related reference:

-  [DCLGEN \(DECLARATIONS GENERATOR\) \(DSN\) \(DB2 Commands\)](#)
-  [DSN \(TSO\) \(DB2 Commands\)](#)
-  [Reserved words \(DB2 SQL\)](#)

Generating table and view declarations by using DCLGEN from DB2I

DCLGEN generates table and view declarations and the corresponding variable declarations for C, COBOL, and PL/I programs so that you do not need to code these statements yourself. The easiest way to start DCLGEN is through DB2I.

To generate table and view declarations by using DCLGEN from DB2I:

1. From the DB2I Primary Option Menu panel, select the **DCLGEN** option. The following DCLGEN panel is displayed:

```

DSNEDP01          DCLGEN          SSID: DSN
====>

Enter table name for which declarations are required:
1 SOURCE TABLE NAME ====>

2 TABLE OWNER ..... ====>

3 AT LOCATION ..... ====> (Optional)
Enter destination data set: (Can be sequential or partitioned)
4 DATA SET NAME ... ====>
5 DATA SET PASSWORD ====> (If password protected)

Enter options as desired:
6 ACTION ..... ====> ADD (ADD new or REPLACE old declaration)
7 COLUMN LABEL ... ====> NO (Enter YES for column label)
8 STRUCTURE NAME .. ====> (Optional)
9 FIELD NAME PREFIX ====> (Optional)
10 DELIMIT DBCS ... ====> YES (Enter YES to delimit DBCS identifiers)
11 COLUMN SUFFIX ... ====> NO (Enter YES to append column name)
12 INDICATOR VARS .. ====> NO (Enter YES for indicator variables)
13 ADDITIONAL OPTIONS====> YES (Enter YES to change additional options)

PRESS: ENTER to process  END to exit  HELP for more information

```

Figure 5. DCLGEN panel

2. Complete the following fields on the DCLGEN panel:

1 SOURCE TABLE NAME

Is the unqualified name of the table, view, or created temporary table for which you want DCLGEN to produce SQL data declarations. The table can be stored at your DB2 location or at another DB2 location. To specify a table name at another DB2 location, enter the table qualifier in the TABLE OWNER field and the location name in the AT LOCATION field. DCLGEN generates a three-part table name from the SOURCE TABLE NAME, TABLE OWNER, and AT LOCATION fields. You can also use an alias for a table name.

To specify a table name that contains special characters or blanks, enclose the name in apostrophes. If the name contains apostrophes, you must double each one(' '). For example, to specify a table named DON'S TABLE, enter the following text:

```
'DON' 'S TABLE'
```

The underscore is not handled as a special character in DCLGEN. For example, the table name JUNE_PROFITS does not need to be enclosed in apostrophes. Because COBOL field names cannot contain underscores, DCLGEN substitutes hyphens (-) for single-byte underscores in COBOL field names that are built from the table name.

You do not need to enclose DBCS table names in apostrophes.

If you do not enclose the table name in apostrophes, DB2 converts lowercase characters to uppercase.

2 TABLE OWNER

Is the schema qualifier of the source table. If you do not specify this value and the table is a local table, DB2 assumes that the table qualifier is your TSO logon ID. If the table is at a remote location, you must specify this value.

3 AT LOCATION

Is the location of a table or view at another DB2 subsystem. The value of the AT LOCATION field becomes a prefix for the table name on the SQL DECLARE statement, as follows: *location_name, schema_name,*

table_name For example, if the location name is PLAINS_GA, the schema name is CARTER, and the table name is CROP_YIELD_89, the following table name is included in the SQL DECLARE statement: PLAINS_GA.CARTER.CROP_YIELD_89

The default is the local location name. This field applies to DB2 private protocol access only. The location must be another DB2 for z/OS subsystem.

4 DATA SET NAME

Is the name of the data set that you allocated to contain the declarations that DCLGEN produces. You must supply a name; no default exists.

The data set must already exist and be accessible to DCLGEN. The data set can be either sequential or partitioned. If you do not enclose the data set name in apostrophes, DCLGEN adds a standard TSO prefix (user ID) and suffix (language). DCLGEN determines the host language from the DB2I defaults panel.

For example, for library name LIBNAME(MEMBNAME), the name becomes *userid.libname.language(membrname)* For library name LIBNAME, the name becomes *userid.libname.language*.

If this data set is password protected, you must supply the password in the DATA SET PASSWORD field.

5 DATA SET PASSWORD

Is the password for the data set that is specified in the DATA SET NAME field, if the data set is password protected. The password is not displayed on your terminal, and it is not recognized if you issued it from a previous session.

6 ACTION

Specifies what DCLGEN is to do with the output when it is sent to a partitioned data set. (The option is ignored if the data set you specify in the DATA SET NAME field is sequential.) You can specify one of the following values:

ADD

Indicates that an old version of the output does not exist and creates a new member with the specified data set name. ADD is the default.

REPLACE

Replaces an old version, if it already exists. If the member does not exist, this option creates a new member.

7 COLUMN LABEL

Specifies whether DCLGEN is to include labels that are declared on any columns of the table or view as comments in the data declarations. (The SQL LABEL statement creates column labels to use as supplements to column names.) You can specify one of the following values:

YES

Include column labels.

NO

Ignore column labels. NO is the default.

8 STRUCTURE NAME

Is the name of the generated data structure. The name can be up to 31

characters. If the name is not a DBCS string, and the first character is not alphabetic, enclose the name in apostrophes. If you use special characters, be careful to avoid name conflicts.

If you leave this field blank, DCLGEN generates a name that contains the table or view name with a prefix of DCL. If the language is COBOL or PL/I and the table or view name consists of a DBCS string, the prefix consists of DBCS characters.

For C, lowercase characters that you enter in this field are not converted to uppercase.

9 FIELD NAME PREFIX

Specifies a prefix that DCLGEN uses to form field names in the output. For example, if you choose ABCDE, the field names generated are ABCDE1, ABCDE2, and so on.

You can specify a field name prefix of up to 28 bytes that can include special and double-byte characters. If you specify a single-byte or mixed-string prefix and the first character is not alphabetic, enclose the prefix in apostrophes. If you use special characters, be careful to avoid name conflicts.

For COBOL and PL/I, if the name is a DBCS string, DCLGEN generates DBCS equivalents of the suffix numbers.

For C, lowercase characters that you enter in this field do not converted to uppercase.

If you leave this field blank, the field names are the same as the column names in the table or view.

10 DELIMIT DBCS

Specifies whether DCLGEN is to delimit DBCS table names and column names in the table declaration. You can specify one of the following values:

YES

Specifies that DCLGEN is to enclose the DBCS table and column names with SQL delimiters.

NO Specifies that DCLGEN is not to delimit the DBCS table and column names.

11 COLUMN SUFFIX

Specifies whether DCLGEN is to form field names by attaching the column name as a suffix to the value that you specify in FIELD NAME PREFIX. You can specify one of the following values:

YES

Specifies that DCLGEN is to use the column name as a suffix. For example, if you specify YES, the field name prefix is NEW, and the column name is EMPNO, the field name is NEWEMPNO.

If you specify YES, you must also enter a value in FIELD NAME PREFIX. If you do not enter a field name prefix, DCLGEN issues a warning message and uses the column names as the field names.

NO Specifies that DCLGEN is not to use the column name as a suffix. The default is NO.

12 INDICATOR VARS

Specifies whether DCLGEN is to generate an array of indicator variables for the host variable structure. You can specify one of the following values:

YES

Specifies that DCLGEN is to generate an array of indicator variables for the host variable structure.

If you specify YES, the array name is the table name with a prefix of I (or DBCS letter <I> if the table name consists solely of double-byte characters). The form of the data declaration depends on the language, as shown in the following table. *n* is the number of columns in the table.

Table 39. Declarations for indicator variable arrays from DCLGEN

Language	Declaration form
C	short int <i>Itable-name</i> [<i>n</i>];
COBOL	01 <i>Itable-name</i> PIC S9(4) USAGE COMP OCCURS <i>n</i> TIMES.
PL/I	DCL <i>Itable-name</i> (<i>n</i>) BIN FIXED(15);

For example, suppose that you define the following table:

```
CREATE TABLE HASNULLS (CHARCOL1 CHAR(1), CHARCOL2 CHAR(1));
```

If you request an array of indicator variables for a COBOL program, DCLGEN might generate the following host variable declaration:

```
01 DCLHASNULLS.  
   10 CHARCOL1          PIC X(1).  
   10 CHARCOL2          PIC X(1).  
01 IHASNULLS PIC S9(4) USAGE COMP OCCURS 2 TIMES.
```

NO Specifies that DCLGEN is not to generate an array of indicator variables. The default is NO.

13 ADDITIONAL OPTIONS

Indicates whether to display the panel for additional DCLGEN options, including the break point for statement tokens and whether to generate DECLARE VARIABLE statements for FOR BIT DATA columns. You can specify YES or NO. The default is YES.

If you specified YES in the ADDITIONAL OPTIONS field, the following ADDITIONAL DCLGEN OPTIONS panel is displayed:

```
DSNEDP02          ADDITIONAL DCLGEN OPTIONS          SSID: DSN  
====>  
  
Enter options as desired:  
1  RIGHT MARGIN .... ==> 72          (Enter 72 or 80)  
  
2  FOR BIT DATA .... ==> NO        (Enter YES to declare SQL variables for  
                                     FOR BIT DATA columns)  
  
PRESS: ENTER to process   END to exit   HELP for more information
```

Figure 6. ADDITIONAL DCLGEN OPTIONS panel

Otherwise, DCLGEN creates the declarations in the specified data set.

3. If the ADDITIONAL DCLGEN OPTIONS panel is displayed, complete the following fields on that panel:

1 RIGHT MARGIN

Specifies the break point for statement tokens that must be wrapped to one or more subsequent records. You can specify column 72 or column 80.

The default is 72.

2 FOR BIT DATA

Specifies whether DCLGEN is to generate a DECLARE VARIABLE statement for SQL variables for columns that are declared as FOR BIT DATA. This statement is required in DB2 applications that meet all of the following criteria:

- are written in COBOL
- have host variables for FOR BIT DATA columns
- are prepared with the SQLCCSID option of the DB2 coprocessor.

You can specify YES or NO. The default is NO.

If the table or view does not have FOR BIT DATA columns, DCLGEN does not generate this statement.

DCLGEN creates the declarations in the specified data set.

Related reference:

“DB2I primary option menu” on page 981

 LABEL (DB2 SQL)

Data types that DCLGEN uses for variable declarations

DCLGEN produces declarations for tables and views and the corresponding host variable structures for C, COBOL, and PL/I programs. DCLGEN derives the variable names and data types for these declarations based on the source tables in the database.

The following table lists the C, COBOL, and PL/I data types that DCLGEN uses for variable declarations based on the corresponding SQL data types that are used in the source tables. *var* represents a variable name that DCLGEN provides.

Table 40. Type declarations that DCLGEN generates

SQL data type ¹	C	COBOL	PL/I
SMALLINT	short int	PIC S9(4) USAGE COMP	BIN FIXED(15)
INTEGER	long int	PIC S9(9) USAGE COMP	BIN FIXED(31)
DECIMAL(p,s) or NUMERIC(p,s)	decimal(p,s) ²	PIC S9(p-s)V9(s) USAGE COMP-3	DEC FIXED(p,s) If p>15, the PL/I compiler must support this precision, or a warning is generated.
REAL or FLOAT(n) 1 <= n <= 21	float	USAGE COMP-1	BIN FLOAT(n)
DOUBLE PRECISION, DOUBLE, or FLOAT(n)	double	USAGE COMP-2	BIN FLOAT(n)
CHAR(1)	char	PIC X(1)	CHAR(1)
CHAR(n)	char var [n+1]	PIC X(n)	CHAR(n)

Table 40. Type declarations that DCLGEN generates (continued)

SQL data type ¹	C	COBOL	PL/I
VARCHAR(n)	struct {short int var_len; char var_data[n]; } var;	10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC X(n).	CHAR(n) VAR
CLOB(n) ³	SQL TYPE IS CLOB_LOCATOR	USAGE SQL TYPE IS CLOB-LOCATOR	SQL TYPE IS CLOB_LOCATOR
GRAPHIC(1)	sqldbchar	PIC G(1)	GRAPHIC(1)
GRAPHIC(n)	sqldbchar var[n+1];	PIC G(n) USAGE DISPLAY-1. ⁴ or PIC N(n). ⁴	GRAPHIC(n)
n > 1			
VARGRAPHIC(n)	struct VARGRAPH {short len; sqldbchar data[n]; } var;	10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC G(n) USAGE DISPLAY-1. ⁴ or 10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC N(n). ⁴	GRAPHIC(n) VAR
DBCLOB(n) ³	SQL TYPE IS DBCLOB_LOCATOR	USAGE SQL TYPE IS DBCLOB-LOCATOR	SQL TYPE IS DBCLOB_LOCATOR
BINARY(n)	SQL TYPE IS BINARY(n)	USAGE SQL TYPE IS BINARY(n)	SQL TYPE IS BINARY(n)
VARBINARY(n)	SQL TYPE IS VARBINARY(n)	USAGE SQL TYPE IS VARBINARY(n)	SQL TYPE IS VARBINARY(n)
BLOB(n) ³	SQL TYPE IS BLOB_LOCATOR	USAGE SQL TYPE IS BLOB-LOCATOR	SQL TYPE IS BLOB_LOCATOR
DATE	char var[11] ⁵	PIC X(10) ⁵	CHAR(10) ⁵
TIME	char var[9] ⁶	PIC X(8) ⁶	CHAR(8) ⁶
TIMESTAMP	char var[27]	PIC X(26)	CHAR(26)
TIMESTAMP(0)	char var[20]	PIC X(19)	CHAR(19)
TIMESTAMP(p) p > 0	char var[21+p]	PIC X(20+p)	CHAR(20+p)
TIMESTAMP(0) WITH TIME ZONE	struct {short int var_len; char var_data[147]; } var;	01 var. 49 var_LEN PIC S9(4) COMP. 49 var_TEXT PIC X(147).	DCL var CHAR(147) VAR;
TIMESTAMP(p) WITH TIME ZONE	struct {short int var_len; char var_data[148 + p]; } var;	01 var. 49 var_LEN PIC S9(4) COMP. 49 var_TEXT PIC X(148 + p).	DCL var CHAR(148 + p) VAR;
ROWID	SQL TYPE IS ROWID	USAGE SQL TYPE IS ROWID	SQL TYPE IS ROWID
BIGINT	long long int	PIC S9(18) USAGE COMP	FIXED BIN(63)
XML ⁷	SQL TYPE IS XML AS CLOB(1M)	SQL TYPE IS XML AS CLOB(1M)	SQL TYPE IS XML AS CLOB(1M)

Table 40. Type declarations that DCLGEN generates (continued)

SQL data type ¹	C	COBOL	PL/I
Notes:			
1. For a distinct type, DCLGEN generates the host language equivalent of the source data type.			
2. If your C compiler does not support the decimal data type, edit your DCLGEN output and replace the decimal data declarations with declarations of type double.			
3. For a BLOB, CLOB, or DBCLOB data type, DCLGEN generates a LOB locator.			
4. DCLGEN chooses the format based on the character that you specify as the DBCS symbol on the COBOL Defaults panel.			
5. This declaration is used unless a date installation exit routine exists for formatting dates, in which case the length is that specified for the LOCAL DATE LENGTH installation option.			
6. This declaration is used unless a time installation exit routine exists for formatting times, in which case the length is that specified for the LOCAL TIME LENGTH installation option.			
7. The default setting for XML is 1M; however, you might need to adjust it.			

Including declarations from DCLGEN in your program

After you use DCLGEN to produce declarations for tables, views, and variables for your C, COBOL, or PL/I program, you should include these declarations in your program.

Recommendation: To ensure that your program uses a current description of the table, use DCLGEN to generate the table's declaration and store it as a member in a library (usually a partitioned data set) just before you precompile the program.

To include declarations from DCLGEN in your program:

Code the following SQL INCLUDE statement in your program:

```
EXEC SQL
    INCLUDE member-name
END-EXEC.
```

member-name is the name of the data set member where the DCLGEN output is stored.

Example: Suppose that you used DCLGEN to generate a table declaration and corresponding COBOL record description for the table DSN8A10.EMP, and those declarations were stored in the data set member DECEMP. (A COBOL record description is a two-level host structure that corresponds to the columns of a table's row.) To include those declarations in your program, include the following statement in your COBOL program:

```
EXEC SQL
    INCLUDE DECEMP
END-EXEC.
```

Related reference:

 [INCLUDE \(DB2 SQL\)](#)

Example: Adding DCLGEN declarations to a library

You can use DCLGEN to generate table and variable declarations for C, COBOL, and PL/I programs. If you store these declarations in a library, you can later integrate them into your program with a single SQL INCLUDE statement.

This example adds a table declaration and a corresponding host-variable structure to a library. This example is based on the following scenario:

- The library name is *prefix*.TEMP.COBOL.
- The member is a new member named VPHONE.
- The table is a local table named DSN8A10.VPHONE.
- The host-variable structure is for COBOL.
- The structure receives the default name DCLVPHONE.

Throughout this example, information that you must enter on each panel is in bold-faced type.

In this scenario, to add a table declaration and a corresponding host variable structure for DSN8A10.VPHONE to the library *prefix*.TEMP.COBOL, complete the following steps:

1. Specify COBOL as the host language by completing the following actions:
 - a. On the ISPF/PDF menu, select option **D** to display the DB2I DEFAULTS PANEL 1 panel.
 - b. Specify **IBMCOB** as the application language, as shown in the following figure and press Enter.

```
DSNEOP01                DB2I DEFAULTS PANEL 1
COMMAND ==>_

Change defaults as desired:

 1 DB2 NAME ..... ==> DSN          (Subsystem identifier)
 2 DB2 CONNECTION RETRIES ==> 0      (How many retries for DB2 connection)
 3 APPLICATION LANGUAGE ==> IBMCOB  (ASM, C, CPP, IBMCOB, FORTRAN, PLI)
 4 LINES/PAGE OF LISTING ==> 80      (A number from 5 to 999)
 5 MESSAGE LEVEL ..... ==> I        (Information, Warning, Error, Severe)
 6 SQL STRING DELIMITER ==> DEFAULT  (DEFAULT, ' or ")
 7 DECIMAL POINT ..... ==> .        (. or ,)
 8 STOP IF RETURN CODE >= ==> 8      (Lowest terminating return code)
 9 NUMBER OF ROWS ..... ==> 20      (For ISPF Tables)
10 CHANGE HELP BOOK NAMES?==> NO    (YES to change HELP data set names)
11 AS USER ..... ==>                (Userid to associate with the trusted
                                     connection)

PRESS: ENTER to process      END to cancel      HELP for more information
```

Figure 7. DB2I defaults panel—changing the application language

The DB2I DEFAULTS PANEL 2 panel for COBOL is then displayed.

- c. Complete the DB2I DEFAULTS PANEL 2 panel, shown in the following figure, as needed and press Enter to save the new defaults, if any.

```

DSNEOP02                DB2I DEFAULTS PANEL 2
COMMAND ==>_

Change defaults as desired:

1 DB2I JOB STATEMENT:  (Optional if your site has a SUBMIT exit)
  ==> //ADMF001A JOB (ACCOUNT),'NAME'
  ==> /**
  ==> /**
  ==> /**

COBOL DEFAULTS:                (For IBMCOB)
2 COBOL STRING DELIMITER ==> DEFAULT (DEFAULT, ' or ")
3 DBCS SYMBOL FOR DCLGEN ==> G      (G/N - Character in PIC clause)

```

Figure 8. The COBOL defaults panel. Shown only if the field APPLICATION LANGUAGE on the DB2I DEFAULTS PANEL I panel is IBMCOB.

The DB2I Primary Option menu is displayed.

2. Generate the table and host structure declarations by completing the following actions:
 - a. On the DB2I Primary Option menu, select the **DCLGEN** option and press Enter to display the DCLGEN panel.
 - b. Complete the fields as shown in the following figure and press Enter.

```

DSNEDP01                DCLGEN                SSID: DSN
==>

Enter table name for which declarations are required:
1 SOURCE TABLE NAME ==> DSN8A10.VPHONE

2 TABLE OWNER ..... ==>

3 AT LOCATION ..... ==>                                (Optional)
Enter destination data set:                               (Can be sequential or partitioned)
4 DATA SET NAME ... ==> TEMP(VPHONEC)
5 DATA SET PASSWORD ==>                                (If password protected)

Enter options as desired:
6 ACTION ..... ==> ADD      (ADD new or REPLACE old declaration)
7 COLUMN LABEL ... ==> NO      (Enter YES for column label)
8 STRUCTURE NAME .. ==>                                (Optional)
9 FIELD NAME PREFIX ==>                                (Optional)
10 DELIMIT DBCS ... ==> YES    (Enter YES to delimit DBCS identifiers)
11 COLUMN SUFFIX ... ==> NO    (Enter YES to append column name)
12 INDICATOR VARS .. ==> NO    (Enter YES for indicator variables)
13 ADDITIONAL OPTIONS==> NO  (Enter YES to change additional options)

PRESS: ENTER to process  END to exit  HELP for more information

```

Figure 9. DCLGEN panel—selecting source table and destination data set

A successful completion message, such as the one in the following figure, is displayed at the top of your screen.

```

DSNE905I EXECUTION COMPLETE, MEMBER VPHONEC ADDED
***

```

Figure 10. Successful completion message

DB2 again displays the DCLGEN screen, as shown in the following figure.

```

DSNEDP01          DCLGEN          SSID: DSN
====>

Enter table name for which declarations are required:
1 SOURCE TABLE NAME ====> DSN8A10.VPHONE

2 TABLE OWNER ..... ====>

3 AT LOCATION ..... ====> (Optional)
Enter destination data set: (Can be sequential or partitioned)
4 DATA SET NAME ... ====> TEMP(VPHONEC)
5 DATA SET PASSWORD ====> (If password protected)
Enter options as desired:
6 ACTION ..... ====> ADD (ADD new or REPLACE old declaration)
7 COLUMN LABEL ... ====> NO (Enter YES for column label)
8 STRUCTURE NAME .. ====> (Optional)
9 FIELD NAME PREFIX ====> (Optional)
10 DELIMIT DBCS ... ====> YES (Enter YES to delimit DBCS identifiers)
11 COLUMN SUFFIX ... ====> NO (Enter YES to append column name)
12 INDICATOR VARS .. ====> NO (Enter YES for indicator variables)
13 ADDITIONAL OPTIONS====> NO (Enter YES to change additional options)

PRESS: ENTER to process  END to exit  HELP for more information

```

Figure 11. DCLGEN panel—displaying system and user return codes

- c. Press Enter to return to the DB2I Primary Option menu.
3. Exit from DB2I.
4. Examine the DCLGEN output by selecting either the browse or the edit option from the ISPF/PDF menu to view the results in the specified data set member. For this example, the data set to edit is *prefix*.TEMP.COBOLE(VPHONEC). This data set member contains the following information.

```

***** DCLGEN TABLE(DSN8A10.VPHONE) *****
***** LIBRARY(SYSADM.TEMP.COBOLE(VPHONEC)) *****
***** QUOTE *****
***** ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS *****
EXEC SQL DECLARE DSN8A10.VPHONE TABLE
( LASTNAME          VARCHAR(15) NOT NULL,
  FIRSTNAME         VARCHAR(12) NOT NULL,
  MIDDLEINITIAL     CHAR(1) NOT NULL,
  PHONENUMBER       VARCHAR(4) NOT NULL,
  EMPLOYEEENUMBER   CHAR(6) NOT NULL,
  DEPTNUMBER        CHAR(3) NOT NULL,
  DEPTNAME          VARCHAR(36) NOT NULL
) END-EXEC.
***** COBOL DECLARATION FOR TABLE DSN8A10.VPHONE *****
01 DCLVPHONE.
  10 LASTNAME.
    49 LASTNAME-LEN      PIC S9(4) USAGE COMP.
    49 LASTNAME-TEXT    PIC X(15).
  10 FIRSTNAME.
    49 FIRSTNAME-LEN    PIC S9(4) USAGE COMP.
    49 FIRSTNAME-TEXT  PIC X(12).
  10 MIDDLEINITIAL     PIC X(1).
  10 PHONENUMBER.
    49 PHONENUMBER-LEN  PIC S9(4) USAGE COMP.
    49 PHONENUMBER-TEXT PIC X(4).
  10 EMPLOYEEENUMBER   PIC X(6).
  10 DEPTNUMBER        PIC X(3).
  10 DEPTNAME.
    49 DEPTNAME-LEN     PIC S9(4) USAGE COMP.
    49 DEPTNAME-TEXT    PIC X(36).
***** THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7 *****

```

You can now pull these declarations into your program by using an SQL INCLUDE statement.

Defining the items that your program can use to check whether an SQL statement executed successfully

If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

Whether you define the SQLCODE or SQLSTATE variables or an SQLCA in your program depends on what you specify for the SQL processing option STDSQL.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

Related tasks:

“Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler” on page 253

“Defining the SQL communications area, SQLSTATE, and SQLCODE in C” on page 273

“Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL” on page 323

“Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran” on page 395

“Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I” on page 407

“Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX” on page 437

Related reference:

“Descriptions of SQL processing options” on page 932

 [Description of SQLCA fields \(DB2 SQL\)](#)

 [INCLUDE \(DB2 SQL\)](#)

 [The REXX SQLCA \(DB2 SQL\)](#)

Defining SQL descriptor areas

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

If your program includes any of the following statements, you must include an SQLDA in your program:

- CALL ... USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR *host-variable* INTO *descriptor-name*
- DESCRIBE INPUT *statement-name* INTO *descriptor-name*
- DESCRIBE PROCEDURE *host-variable* INTO *descriptor-name*
- DESCRIBE TABLE *host-variable* INTO *descriptor-name*
- EXECUTE ... USING DESCRIPTOR *descriptor-name*

- FETCH ... INTO DESCRIPTOR *descriptor-name*
- OPEN ... USING DESCRIPTOR *descriptor-name*
- PREPARE ... INTO *descriptor-name*

Unlike the SQLCA, a program can have more than one SQLDA, and an SQLDA can have any valid name.

To define SQL descriptor areas:

Take the actions that are appropriate for the programming language that you use.

Related tasks:

“Defining SQL descriptor areas in assembler” on page 254

“Defining SQL descriptor areas in C” on page 274

“Defining SQL descriptor areas in COBOL” on page 324

“Defining SQL descriptor areas in Fortran” on page 396

“Defining SQL descriptor areas in PL/I” on page 408

“Defining SQL descriptor areas in REXX” on page 437

Related reference:

“Descriptions of SQL processing options” on page 932

 Description of SQLCA fields (DB2 SQL)

 SQL descriptor area (SQLDA) (DB2 SQL)

 The REXX SQLCA (DB2 SQL)

Declaring host variables and indicator variables

You can use host variables and indicator variables in SQL statements in your program to pass data between DB2 and your application.

To declare host variables, host variable arrays, and host structures:

Use the techniques that are appropriate for the programming language that you use.

Related tasks:

“Accessing data by using a rowset-positioned cursor” on page 731

“Determining whether a retrieved value in a host variable is null or truncated” on page 186

Related reference:

“Descriptions of SQL processing options” on page 932

Host variables

Use host variables to pass a single data item between DB2 and your application.

A *host variable* is a single data item that is declared in the host language to be used within an SQL statement. You can use host variables in application programs that are written in the following languages: assembler, C, C++, COBOL, Fortran, and PL/I to perform the following actions:

- Retrieve data into the host variable for your application program's use
- Place data into the host variable to insert into a table or to change the contents of a row

- Use the data in the host variable when evaluating a WHERE or HAVING clause
- Assign the value that is in the host variable to a special register, such as CURRENT SQLID and CURRENT DEGREE
- Insert null values into columns by using a host indicator variable that contains a negative value
- Use the data in the host variable in statements that process dynamic SQL, such as EXECUTE, PREPARE, and OPEN

Related concepts:

“Rules for host variables in an SQL statement” on page 183

Related reference:

“Host variables in assembler” on page 255

“Host variables in C” on page 275

“Host variables in COBOL” on page 326

“Host variables in Fortran” on page 397

“Host variables in PL/I” on page 409

Host variable arrays

Use host variable arrays to pass a data array between DB2 and your application.

A *host variable array* is a data array that is declared in the host language to be used within an SQL statement. You can use host variable arrays to perform the following actions:

- Retrieve data into host variable arrays for your application program's use
- Place data into host variable arrays to insert rows into a table

You typically define host variable arrays for use with multiple-row FETCH, INSERT, and MERGE statements.

Related concepts:

“Host variable arrays in an SQL statement” on page 191

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

“Retrieving multiple rows of data into host variable arrays” on page 192

Related reference:

“Host variable arrays in C” on page 287

“Host variable arrays in COBOL” on page 335

“Host variable arrays in PL/I” on page 415

Host structures

Use host structures to pass a group of host variables between DB2 and your application.

A *host structure* is a group of host variables that can be referenced with a single name. You can use host structures in all host languages except REXX. You define host structures with statements in the host language. You can refer to a host structure in any context where you want to refer to the list of host variables in the structure. A host structure reference is equivalent to a reference to each of the host variables within the structure in the order in which they are defined in the structure declaration. You can also use indicator variables (or indicator structures) with host structures.

Related tasks:

“Retrieving a single row of data into a host structure” on page 193

Related reference:

“Host structures in C” on page 295

“Host structures in COBOL” on page 344

“Host structures in PL/I” on page 420

Indicator variables, arrays, and structures

An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host variable arrays and structures.

You can use indicator variables to perform the following actions:

- Determine whether the value of an associated output host variable is null or indicate that the value of an input host variable is null
- Determine the original length of a character string that was truncated when it was assigned to a host variable
- Determine that a character value could not be converted when it was assigned to a host variable
- Determine the seconds portion of a time value that was truncated when it was assigned to a host variable
- Indicate that the target column of the host variable is to be set to its defined DEFAULT value, or that the host variable's value is UNASSIGNED and its target column is to be treated as if it had not appeared in the statement.

You can use indicator variable arrays and indicator structures to perform these same actions for individual items in host data arrays and structures.

If you provide an indicator variable for the variable X, when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, you know that X is null and any value that you find in X is irrelevant. When your program uses variable X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

An indicator variable array contains a series of small integers to help you determine the associated information for the corresponding item in a host data array. When you retrieve data into a host variable array, you can check the values in the associated indicator array to determine how to handle each data item. If a value in the associated indicator array is negative, you can disregard the contents of the corresponding element in the host variable array. Values in indicator arrays have the following meanings:

On output to the application, the normal indicator variable can contain the following values:

- 0** A 0 (zero), or positive value of the indicator variable specifies that the first host-identifier provides the value of this host variable reference.
- 1** A -1 value indicates that the value that was selected was the null value.
- 2** A -2 value of the indicator variable indicates that a numeric conversion error (such as a divide by 0 or overflow) has occurred. Or indicates a null result because of character string conversion warnings.

-3 A -3 value of the indicator variable indicates that no value was returned. A -3 value of the indicator variable can also indicate a null result because the cursor's current row is on a hole that was detected during a multiple row FETCH.

positive integer

If the indicator variable contains a positive integer, the retrieved value is truncated, and the integer is the original length of the string.

positive integer

The seconds portion of a time if the time is truncated on assignment to a host variable.

On input to DB2, normal indicator variables or extended indicator variables can contain the following values:

0, or positive integer

Specifies a non-null value. A 0 (zero), or positive value of the indicator variable specifies that the first host-identifier provides the value of this host variable reference.

-1, -2, -3, -4, -6

Specifies a null value.

-5

- If the extended indicator variable is not enabled, a -5 value specifies the NULL value.
- If the extended indicator variable is enabled, a -5 value specifies the DEFAULT value. A -5 value specifies that the target column for this host variable is to be set to its DEFAULT value.

-7

- If the extended indicator variable is not enabled, a -7 value specifies the NULL value.
- If the extended indicator variable is enabled, a -7 value specifies the an UNASSIGNED value. A -7 value specifies that the target column for this host variable is to be treated as if it hadn't been specified in the statement.

An *indicator structure* is an array of halfword integer variables that supports a specified host structure. If the column values that your program retrieves into a host structure can be null, you can attach an indicator structure name to the host structure name. This name enables DB2 to notify your program about each null value it returns to a host variable in the host structure.

Related concepts:

"Holes in the result table of a scrollable cursor" on page 741

Related tasks:

"Executing SQL statements by using a rowset cursor" on page 733

Related reference:

"Indicator variables in assembler" on page 260

"Indicator variables, indicator arrays, and host structure indicator arrays in C" on page 297

"Indicator variables, indicator arrays, and host structure indicator arrays in COBOL" on page 349

"Indicator variables in Fortran" on page 400

“Indicator variables in PL/I” on page 422

Setting the CCSID for host variables

All DB2 string data, other than binary data, has an encoding scheme and a coded character set ID (CCSID) associated with it. You can associate an encoding scheme and a CCSID with individual host variables. Any data in those host variable is then associated with that encoding scheme and CCSID.

To set the CCSID for host variables:

Specify the DECLARE VARIABLE statement after the corresponding host variable declaration and before your first reference to that host variable.

This statement associates an encoding scheme and a CCSID with individual host variables. You can use this statement in static or dynamic SQL applications.

Restriction: You cannot use the DECLARE VARIABLE statement to control the CCSID and encoding scheme of data that you retrieve or update by using an SQLDA.

The DECLARE VARIABLE statement has the following effects on a host variable:

- When you use the host variable to update a table, the local subsystem or the remote server assumes that the data in the host variable is encoded with the CCSID and encoding scheme that the DECLARE VARIABLE statement assigns.
- When you retrieve data from a local or remote table into the host variable, the retrieved data is converted to the CCSID and encoding scheme that are assigned by the DECLARE VARIABLE statement.

Suppose that you are writing a C program that runs on a DB2 for z/OS subsystem. The subsystem has an EBCDIC application encoding scheme. The C program retrieves data from the following columns of a local table that is defined with the CCSID UNICODE option:

```
PARTNUM CHAR(10)
JPNNAME GRAPHIC(10)
ENGNAME VARCHAR(30)
```

Because the application encoding scheme for the subsystem is EBCDIC, the retrieved data is EBCDIC. To make the retrieved data Unicode, use DECLARE VARIABLE statements to specify that the data that is retrieved from these columns is encoded in the default Unicode CCSIDs for the subsystem.

Suppose that you want to retrieve the character data in Unicode CCSID 1208 and the graphic data in Unicode CCSID 1200. Use the following DECLARE VARIABLE statements:

```
EXEC SQL BEGIN DECLARE SECTION;
char hvpartnum[11];
EXEC SQL DECLARE :hvpartnum VARIABLE CCSID 1208;
sqldbcchar hvjpnnname[11];
EXEC SQL DECLARE :hvjpnnname VARIABLE CCSID 1200;
struct {
    short len;
    char d[30];
} hvengname;
EXEC SQL DECLARE :hvengname VARIABLE CCSID 1208;
EXEC SQL END DECLARE SECTION;
```

Related reference:

 [DECLARE VARIABLE \(DB2 SQL\)](#)

Determining what caused an error when retrieving data into a host variable

Errors that occur when DB2 passes data to host variables in an application are usually caused by a problem in converting from one data type to another. These errors do not affect the position of the cursor.

For example, suppose that you fetch an integer value of 32768 into a host variable of type SMALLINT. The conversion might cause an error if you do not provide sufficient conversion information to DB2.

The variable to which DB2 assigns the data is called the *output host variable*. If you provide an indicator variable for the output host variable or if data type conversion is not required, DB2 returns a positive SQLCODE for the row in most cases. In other cases where data conversion problems occur, DB2 returns a negative SQLCODE for that row. Regardless of the SQLCODE for the row, no new values are assigned to the host variable or to subsequent variables for that row. Any values that are already assigned to variables remain assigned. Even when a negative SQLCODE is returned for a row, statement processing continues and DB2 returns a positive SQLCODE for the statement (SQLSTATE 01668, SQLCODE +354).

To determine what caused an error when retrieving data into a host variable:

1. When DB2 returns SQLCODE = +354, use the GET DIAGNOSTICS statement with the NUMBER option to determine the number of errors and warnings.

Example: Suppose that no indicator variables are provided for the values that are returned by the following statement:

```
FETCH FIRST ROWSET FROM C1 FOR 10 ROWS INTO :hva_col1, :hva_col2;
```

For each row with an error, DB2 records a negative SQLCODE and continues processing until the 10 rows are fetched. When SQLCODE = +354 is returned for the statement, you can use the GET DIAGNOSTICS statement to determine which errors occurred for which rows. The following statement returns num_rows = 10 and num_cond = 3:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

2. To investigate the errors and warnings, use additional GET DIAGNOSTIC statements with the CONDITION option.

Example: To investigate the three conditions that were reported in the example in the previous step, use the following statements:

Table 41. GET DIAGNOSTIC statements to investigate conditions

Statement	Output
GET DIAGNOSTICS CONDITION 3 :sqlstate = RETURNED_SQLSTATE, :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;	sqlstate = 22003 sqlcode = -304 row_num = 5
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;	sqlstate = 22003 sqlcode = -802 row_num = 7

Table 41. GET DIAGNOSTIC statements to investigate conditions (continued)

Statement	Output
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;	sqlstate = 01668 sqlcode = +354 row_num = 0

This output shows that the fifth row has a data mapping error (-304) for column 1 and that the seventh row has a data mapping error (-802) for column 2. These rows do not contain valid data, and they should not be used.

Related concepts:

“Indicator variables, arrays, and structures” on page 176

Related reference:

 GET DIAGNOSTICS (DB2 SQL)

Related information:

 +354 (DB2 Codes)

Accessing an application defaults module

If your application program currently uses LOAD DSNHDECP, consider changing the application program to use the DECP address that is returned by ICFID 373, DSNALI, or DSNRLI.

By using the DECP address that is returned by IFCID 373, DSNALI, or DSNRLI, guarantees that you are using the same DECP module that was used to start DB2. It also allows the code to skip the LOAD entirely, only after successfully connecting to DB2. DSNHDECP is loaded by DB2 into Global, pageable storage, so all programs can share it.

Compatibility of SQL and language data types

The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

When deciding the data types of host variables, consider the following rules and recommendations:

- Numeric data types are compatible with each other:
 - Assembler:** A SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT column is compatible with a numeric assembler host variable.
 - Fortran:** An INTEGER column is compatible with any Fortran host variable that is defined as INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, or DOUBLE PRECISION.
 - PL/I:** A SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT column is compatible with a PL/I host variable of BIN FIXED(15), BIN FIXED(31), DECIMAL(s,p), or BIN FLOAT(n), where n is from 1 to 53, or DEC FLOAT(m) where m is from 1 to 16.
- Character data types are compatible with each other:
 - Assembler:** A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length assembler character host variable.
 - C/C++:** A CHAR, VARCHAR, or CLOB column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.

COBOL: A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length COBOL character host variable.

Fortran: A CHAR, VARCHAR, or CLOB column is compatible with Fortran character host variable.

PL/I: A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length PL/I character host variable.

- Character data types are partially compatible with CLOB locators. You can perform the following assignments:
 - Assign a value in a CLOB locator to a CHAR or VARCHAR column
 - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
 - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
 - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
 - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.

- Graphic data types are compatible with each other:

Assembler: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length assembler graphic character host variable.

C/C++: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a single character, NUL-terminated, or VARGRAPHIC structured form of a C graphic host variable.

COBOL: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length COBOL graphic string host variable.

PL/I: A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length PL/I graphic character host variable.

- Graphic data types are partially compatible with DBCLOB locators. You can perform the following assignments:
 - Assign a value in a DBCLOB locator to a GRAPHIC or VARGRAPHIC column
 - Use a SELECT INTO statement to assign a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
 - Assign a GRAPHIC or VARGRAPHIC output parameter from a user-defined function or stored procedure to a DBCLOB locator host variable.
 - Use a SET assignment statement to assign a GRAPHIC or VARGRAPHIC transition variable to a DBCLOB locator host variable.
 - Use a VALUES INTO statement to assign a GRAPHIC or VARGRAPHIC function parameter to a DBCLOB locator host variable.

However, you cannot use a FETCH statement to assign a value in a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.

- Binary data types are compatible with each other.
- Binary data types are partially compatible with BLOB locators. You can perform the following assignments:
 - Assign a value in a BLOB locator to a BINARY or VARBINARY column.
 - Use a SELECT INTO statement to assign a BINARY or VARBINARY column to a BLOB locator host variable.

- Assign a BINARY or VARBINARY output parameter from a user-defined function or stored procedure to a BLOB locator host variable.
- Use a SET assignment statement to assign a BINARY or VARBINARY transition variable to a BLOB locator host variable.
- Use a VALUES INTO statement to assign a BINARY or VARBINARY function parameter to a BLOB locator host variable.

However, you cannot use a FETCH statement to assign a value in a BINARY or VARBINARY column to a BLOB locator host variable.

- **Fortran:** A BINARY, VARBINARY, or BLOB column or BLOB locator is compatible only with a BLOB host variable.
- **C:** For varying-length BIT data, use BINARY. Some C string manipulation functions process NUL-terminated strings and other functions process strings that are not NUL-terminated. The C string manipulation functions that process NUL-terminated strings cannot handle bit data because these functions might misinterpret a NUL character to be a NUL-terminator.
- Datetime data types are compatible with character host variables.
 - Assembler:** A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length assembler character host variable.
 - C/C++:** A DATE, TIME, or TIMESTAMP column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.
 - COBOL:** A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying length COBOL character host variable.
 - Fortran:** A DATE, TIME, or TIMESTAMP column is compatible with a Fortran character host variable.
 - PL/I:** A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.
- The ROWID column is compatible only with a ROWID host variable.
- A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type.
- XML columns are compatible with the XML host variable types, character types, and binary string types.

Recommendation: Use the XML host variable types for data from XML columns.

- **Assembler:** You can assign LOB data to a file reference variable (BLOB_FILE, CLOB_FILE, and DBCLOB_FILE).

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Related concepts:

“Distinct types” on page 513

“Host variable data types for XML data in embedded SQL applications” on page 241

Related reference:

“Equivalent SQL and assembler data types” on page 261

“Equivalent SQL and C data types” on page 301

“Equivalent SQL and COBOL data types” on page 352

“Equivalent SQL and Fortran data types” on page 401

“Equivalent SQL and PL/I data types” on page 423

Embedding SQL statements in your application

You can code SQL statements in an assembler, C, C++, COBOL, Fortran, or PL/I program or REXX procedure wherever you can use executable statements.

To embed SQL statements in your application:

Take action based on the program language that you use.

Related concepts:

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Delimiting an SQL statement

You must delimit SQL statements in your program so that DB2 knows when a particular SQL statement ends.

To delimit an SQL statement:

Take action based on the programming language that you use.

Related concepts:

“Delimiters in SQL statements in assembler programs” on page 271

“Delimiters in SQL statements in C programs” on page 311

“Delimiters in SQL statements in COBOL programs” on page 363

“Delimiters in SQL statements in Fortran programs” on page 406

“Delimiters in SQL statements in PL/I programs” on page 432

“Delimiters in SQL statements in REXX programs” on page 442

Rules for host variables in an SQL statement

Use host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

When you use host variables, adhere to the following requirements:

- You must declare the name of the host variable in the host program before you use it. Host variables follow the naming conventions of the host language.
- You can use a host variable to represent a data value, but you cannot use it to represent a table, view, or column name. You can specify table, view, or column names at run time by using dynamic SQL.
- To use a host variable in an SQL statement, you can specify any valid host variable name that is declared according to the rules of the host language.
- A colon (:) must precede host variables that are used in SQL statements so that DB2 can distinguish a variable name from a column name. When host variables are used outside of SQL statements, do not precede them with a colon. PL/I

programs have the following exceptions: If the SQL statement meets any of the following conditions, do not precede a host variable or host variable array in that statement with a colon:

- The SQL statement is in a program that also contains a DECLARE VARIABLE statement.
- The host variable is part of a string expression, but the host variable is not the only component of the string expression.
- To optimize performance, make sure that the host language declaration maps as closely as possible to the data type of the associated data in the database.
- For assignments and comparisons between a DB2 column and a host variable of a different data type or length, expect conversions to occur.

Related concepts:

 Assignment and comparison (DB2 SQL)

Related tasks:

“Including dynamic SQL in your program” on page 193

Retrieving a single row of data into host variables

If you know that your query returns only one row, you can specify one or more host variables to contain the column values of the retrieved row.

Restriction: These instructions do not apply if you do not know how many rows DB2 will return or if you expect DB2 to return more than one row. In these situations, use a cursor. A cursor enables an application to return a set of rows and fetch either one row at a time or one rowset at a time from the result table.

To retrieve a single row of data into host variables:

In the SELECT statement specify the INTO clause with the name of one or more host variables to contain the retrieved values. Specify one variable for each value that is to be retrieved. The retrieved value can be a column value, a value of a host variable, the result of an expression, or the result of an aggregate function.

Recommendation: If you want to ensure that only one row is returned, specify the FETCH FIRST 1 ROW ONLY clause. Consider using the ORDER BY clause to control which row is returned. If you specify both the ORDER BY clause and the FETCH FIRST clause, ordering is performed on the entire result set before the first row is returned.

DB2 assigns the first value in the result row to the first variable in the list, the second value to the second variable, and so on.

If the SELECT statement returns more than one row, DB2 returns an error, and any data that is returned is undefined and unpredictable.

Examples

Example of retrieving a single row into a host variable: Suppose that you are retrieving the LASTNAME and WORKDEPT column values from the DSN8A10.EMP table for a particular employee. You can define a host variable in your program to hold each column value and then name the host variables in the INTO clause of the SELECT statement, as shown in the following COBOL example.

```
MOVE '000110' TO CBLEMPNO.  
EXEC SQL  
    SELECT LASTNAME, WORKDEPT
```

```

        INTO :CBLNAME, :CBLDEPT
        FROM DSN8A10.EMP
        WHERE EMPNO = :CBLEMPNO
END-EXEC.

```

In this example, the host variable CBLEMPNO is preceded by a colon (:) in the SQL statement, but it is not preceded by a colon in the COBOL MOVE statement.

This example also uses a host variable to specify a value in a search condition. The host variable CBLEMPNO is defined for the employee number, so that you can retrieve the name and the work department of the employee whose number is the same as the value of the host variable, CBLEMPNO; in this case, 000110.

In the DATA DIVISION section of a COBOL program, you must declare the host variables CBLEMPNO, CBLNAME, and CBLDEPT to be compatible with the data types in the columns EMPNO, LASTNAME, and WORKDEPT of the DSN8A10.EMP table.

Example of ensuring that a query returns only a single row: You can use the FETCH FIRST 1 ROW ONLY clause in a SELECT statement to ensure that only one row is returned. This action prevents undefined and unpredictable data from being returned when you specify the INTO clause of the SELECT statement. The following example SELECT statement ensures that only one row of the DSN8A10.EMP table is returned.

```

EXEC SQL
  SELECT LASTNAME, WORKDEPT
  INTO :CBLNAME, :CBLDEPT
  FROM DSN8A10.EMP
  FETCH FIRST 1 ROW ONLY
END-EXEC.

```

You can include an ORDER BY clause in the preceding example to control which row is returned. The following example SELECT statement ensures that the only row returned is the one with a last name that is first alphabetically.

```

EXEC SQL
  SELECT LASTNAME, WORKDEPT
  INTO :CBLNAME, :CBLDEPT
  FROM DSN8810.EMP
  ORDER BY LASTNAME
  FETCH FIRST 1 ROW ONLY
END-EXEC.

```

Example of retrieving the results of host variable values and expressions into host variables:

When you specify a list of items in the SELECT clause, that list can include more than the column names of tables and views. You can request a set of column values mixed with host variable values and constants. For example, the following query requests the values of several columns (EMPNO, LASTNAME, and SALARY), the value of a host variable (RAISE), and the value of the sum of a column and a host variable (SALARY and RAISE). For each of these five items in the SELECT list, a host variable is listed in the INTO clause.

```

MOVE 4476 TO RAISE.
MOVE '000220' TO PERSON.
EXEC SQL
  SELECT EMPNO, LASTNAME, SALARY, :RAISE, SALARY + :RAISE

```

```

        INTO :EMP-NUM, :PERSON-NAME, :EMP-SAL, :EMP-RAISE, :EMP-TTL
        FROM DSN8A10.EMP
        WHERE EMPNO = :PERSON
    END-EXEC.

```

The preceding SELECT statement returns the following results. The column headings represent the names of the host variables.

```

EMP-NUM    PERSON-NAME    EMP-SAL    EMP-RAISE    EMP-TTL
=====    =====    =====    =====    =====
000220    LUTZ            29840      4476         34316

```

Example of retrieving the result of an aggregate function into a host variable: A query can request summary values to be returned from aggregate functions and store those values in host variables. For example, the following query requests that the result of the AVG function be stored in the AVG-SALARY host variable.

```

MOVE 'D11' TO DEPTID.
EXEC SQL
    SELECT WORKDEPT, AVG(SALARY)
    INTO :WORK-DEPT, :AVG-SALARY
    FROM DSN8A10.EMP
    WHERE WORKDEPT = :DEPTID
END-EXEC.

```

Related tasks:

“Retrieving a set of rows by using a cursor” on page 722

Related reference:

 SELECT INTO (DB2 SQL)

Determining whether a retrieved value in a host variable is null or truncated

Before your application manipulates the data that was retrieved from DB2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

Before you determine whether a retrieved column value is null or truncated, you must have defined the appropriate indicator variables, arrays, and structures.

An error occurs if you do not use an indicator variable and DB2 retrieves a null value.

To determine whether a retrieved value in a host variable is null or truncated:

Determine the value of the indicator variable, array, or structure that is associated with the host variable, array, or structure. Those values have the following meanings:

Table 42. Meanings of values in indicator variables

Value of indicator variable	Meaning
Less than zero	The column value is null. The value of the host variable does not change from its previous value.
	If the indicator variable value is -2, the column value is null because of a numeric or character conversion error,

Table 42. Meanings of values in indicator variables (continued)

Value of indicator variable	Meaning
Zero	The column value is nonnull. If the column value is a character string, the retrieved value is not truncated.
Positive integer	The retrieved value is truncated. The integer is the original length of the string.

Examples

Example of testing an indicator variable: Assume that you have defined the following indicator variable INDNULL for the host variable CBLPHONE.

```
EXEC SQL
  SELECT PHONENO
     INTO :CBLPHONE:INDNULL
     FROM DSN8A10.EMP
     WHERE EMPNO = :EMPID
END-EXEC.
```

You can then test INDNULL for a negative value. If the value is negative, the corresponding value of PHONENO is null, and you can disregard the contents of CBLPHONE.

Example of testing an indicator variable array: Suppose that you declare the following indicator array INDNULL for the host variable array CBLPHONE.

```
EXEC SQL
  FETCH NEXT ROWSET CURS1
  FOR 10 ROWS
  INTO :CBLPHONE :INDNULL
END-EXEC.
```

After the multiple-row FETCH statement, you can test each element of the INDNULL array for a negative value. If an element is negative, you can disregard the contents of the corresponding element in the CBLPHONE host variable array.

Example of testing an indicator structure in COBOL: The following example defines the indicator structure EMP-IND as an array that contains six values and corresponds to the PEMP-ROW host structure.

```
01 PEMP-ROW.
  10 EMPNO          PIC X(6).
  10 FIRSTNAME.
    49 FIRSTNAME-LEN PIC S9(4) USAGE COMP.
    49 FIRSTNAME-TEXT PIC X(12).
  10 MIDINIT        PIC X(1).
  10 LASTNAME.
    49 LASTNAME-LEN  PIC S9(4) USAGE COMP.
    49 LASTNAME-TEXT PIC X(15).
  10 WORKDEPT       PIC X(3).
  10 EMP-BIRTHDATE  PIC X(10).
01 INDICATOR-TABLE.
  02 EMP-IND        PIC S9(4) COMP OCCURS 6 TIMES.
:
:
MOVE '000230' TO EMPNO.
:
EXEC SQL
  SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME, WORKDEPT, BIRTHDATE
```



```

        INTO :PEMP-ROW:EMP-IND
        FROM DSN8A10.EMP
        WHERE EMPNO = :EMPNO
    END-EXEC.

```

You can test the indicator structure EMP-IND for negative values. If, for example, EMP-IND(6) contains a negative value, the corresponding host variable in the host structure (EMP-BIRTHDATE) contains a null value.

Related concepts:

“Arithmetic and conversion errors” on page 240

Related tasks:

“Declaring host variables and indicator variables” on page 174

Determining whether a column value is null

Before you retrieve a column value, you might first want to determine if the column value is null.

To determine whether a column value is null:

Use the IS NULL predicate or the IS DISTINCT FROM predicate.

Restriction: You cannot determine whether a column value is null by comparing it to a host variable with an indicator variable that is set to -1.

The following code, which uses an indicator variable, does not select the employees who have no phone number:

```

MOVE -1 TO PHONE-IND.
EXEC SQL
    SELECT LASTNAME
        INTO :PGM-LASTNAME
        FROM DSN8A10.EMP
        WHERE PHONENO = :PHONE-HV:PHONE-IND
    END-EXEC.

```

Instead, use the following statement with the IS NULL predicate to select employees who have no phone number:

```

EXEC SQL
    SELECT LASTNAME
        INTO :PGM-LASTNAME
        FROM DSN8A10.EMP
        WHERE PHONENO IS NULL
    END-EXEC.

```

To select employees whose phone numbers are equal to the value of :PHONE-HV and employees who have no phone number (as in the second example), code two predicates, one to handle the non-null values and another to handle the null values, as in the following statement:

```

EXEC SQL
    SELECT LASTNAME
        INTO :PGM-LASTNAME
        FROM DSN8A10.EMP
        WHERE (PHONENO = :PHONE-HV AND PHONENO IS NOT NULL AND :PHONE-HV IS NOT NULL)
            OR
            (PHONENO IS NULL AND :PHONE-HV:PHONE-IND IS NULL)
    END-EXEC.

```

You can simplify the preceding example by coding the following statement with the NOT form of the IS DISTINCT FROM predicate:

```

EXEC SQL
  SELECT LASTNAME
  INTO :PGM-LASTNAME
  FROM DSN8A10.EMP
  WHERE PHONENO IS NOT DISTINCT FROM :PHONE-HV:PHONE-IND
END-EXEC.

```

Related tasks:

“Declaring host variables and indicator variables” on page 174

Related reference:

 [DISTINCT predicate \(DB2 SQL\)](#)

 [NULL predicate \(DB2 SQL\)](#)

Updating data by using host variables

When you want to update a value in a DB2 table, but you do not know the exact value until the program runs, use host variables. DB2 can change a table value to match the current value of the host variable.

To update data by using host variables:

1. Declare the necessary host variables.
2. Specify an UPDATE statement with the appropriate host variable names in the SET clause.

Examples

Example of updating a single row by using a host variable: The following COBOL example changes an employee's phone number to the value in the NEWPHONE host variable. The employee ID value is passed through the EMPID host variable.

```

MOVE '4246' TO NEWPHONE.
MOVE '000110' TO EMPID.
EXEC SQL
  UPDATE DSN8A10.EMP
  SET PHONENO = :NEWPHONE
  WHERE EMPNO = :EMPID
END-EXEC.

```

Example of updating multiple rows by using a host variable value in the search condition: The following example gives the employees in a particular department a salary increase of 10%. The department value is passed through the DEPTID host variable.

```

MOVE 'D11' TO DEPTID.
EXEC SQL
  UPDATE DSN8A10.EMP
  SET SALARY = 1.10 * SALARY
  WHERE WORKDEPT = :DEPTID
END-EXEC.

```

Related reference:

 [UPDATE \(DB2 SQL\)](#)

Inserting a single row by using a host variable

Use host variables in your INSERT statement when you don't know at least some of the values to insert until the program runs.

Restriction: These instructions apply only to inserting a single row. If you want to insert multiple rows, use host variable arrays or the form of the INSERT statement that selects values from another table or view.

To insert a single row by using host variables:

Specify an INSERT statement with column values in the VALUES clause. Specify host variables or a combination of host variables and constants as the column values.

DB2 inserts the first value into the first column in the list, the second value into the second column, and so on.

The following example uses host variables to insert a single row into the activity table.

```
EXEC SQL
  INSERT INTO DSN8A10.ACT
    VALUES (:HV-ACTNO, :HV-ACTKWD, :HV-ACTDESC)
END-EXEC.
```

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

Related reference:

 INSERT (DB2 SQL)

Inserting null values into columns by using indicator variables or arrays

If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or array.

To insert null values into columns by using indicator variables or arrays:

1. Define an indicator variable or array for a particular host variable or array.
2. Assign a negative value to the indicator variable or array.
3. Issue the appropriate INSERT, UPDATE, or MERGE statement with the host variable or array and its indicator variable or array.

When DB2 processes INSERT, UPDATE, and MERGE statements, it checks the indicator variable if one exists. If the indicator variable is negative, the column value is null. If the indicator variable is greater than -1, the associated host variable contains a value for the column.

Examples

Example of setting a column value to null by using an indicator

variable: Suppose your program reads an employee ID and a new phone number and must update the employee table with the new number. The new number could be missing if the old number is incorrect, but a new number is not yet available. If the new value for column PHONENO might be null, you can use an indicator variable, as shown in the following UPDATE statement.

```
EXEC SQL
  UPDATE DSN8A10.EMP
    SET PHONENO = :NEWPHONE:PHONEIND
    WHERE EMPNO = :EMPID
END-EXEC.
```

When NEWPHONE contains a non-null value, set the indicator variable PHONEIND to zero by preceding the UPDATE statement with the following line:
MOVE 0 TO PHONEIND.

When NEWPHONE contains a null value, set PHONEIND to a negative value by preceding the UPDATE statement with the following line:
MOVE -1 TO PHONEIND.

Example of setting a column value to null by using an indicator variable

array: Assume that host variable arrays hva1 and hva2 have been populated with values that are to be inserted into the ACTNO and ACTKWD columns. Assume the ACTDESC column allows nulls. To set the ACTDESC column to null, assign -1 to the elements in its indicator array, ind3, as shown in the following example:

```
/* Initialize each indicator array */
for (i=0; i<10; i++) {
    ind1[i] = 0;
    ind2[i] = 0;
    ind3[i] = -1;
}

EXEC SQL
  INSERT INTO DSN8A10.ACT
    (ACTNO, ACTKWD, ACTDESC)
    VALUES (:hva1:ind1, :hva2:ind2, :hva3:ind3)
    FOR 10 ROWS;
```

DB2 ignores the values in the hva3 array and assigns the values in the ARTDESC column to null for the 10 rows that are inserted.

Related tasks:

“Declaring host variables and indicator variables” on page 174

Host variable arrays in an SQL statement

Use host variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

To use a host variable array in an SQL statement, specify any valid host variable array that is declared according to the host language rules. You can specify host variable arrays in C or C++, COBOL, and PL/I. You must declare the array in the host program before you use it.

Restrictions: Use of host variable arrays in assembler programs is limited in the following:

- The DB2 precompiler does not recognize declarations of host variable arrays for assembler, it recognizes these declarations only in C, COBOL, and PL/I.
- Assembler does not support multiple-row MERGE. You cannot specify MERGE statements that reference host variable arrays.
- Assembler support for multiple-row FETCH is limited to the FETCH statement with the INTO DESCRIPTOR clause. For example:

```
EXEC SQL FETCH NEXT ROWSET FROM C1 FOR 10 ROWS          X
          INTO DESCRIPTOR :SQLDA
```
- Assembler support for multiple-row INSERT is limited to the following cases:
 - Static multiple-row INSERT statement with scalar values (scalar host variables or scalar expressions) in the VALUES clause. For example:

```
EXEC SQL INSERT INTO T1 VALUES (1, CURRENT DATE, 'TEST') X
FOR 10 ROWS
```

- Dynamic multiple-row INSERT executed with the USING DESCRIPTOR clause on the EXECUTE statement. For example:

```
ATR      DS      CL20                                ATTRIBUTES FOR PREPARE
S1       DS      H,CL30                              VARCHAR STATEMENT STRING
MVC     ATR(20),=C'FOR MULTIPLE ROWS '
MVC     S1(2),=H'25'
MVC     S1+2(30),=C'INSERT INTO T1 VALUES (?) '
EXEC    SQL PREPARE STMT ATTRIBUTES :ATR FROM :S1
EXEC    SQL EXECUTE STMT USING DESCRIPTOR :SQLDA FOR 10 ROWS
```

where the descriptor is set up correctly in advance according to the specifications for dynamic execution of a multiple-row INSERT statement with a descriptor

Related concepts:

“Host variable arrays” on page 175

Related tasks:

“Embedding SQL statements in your application” on page 183

“Inserting multiple rows of data from host variable arrays”

“Retrieving multiple rows of data into host variable arrays”

Retrieving multiple rows of data into host variable arrays

If you know that your query returns multiple rows, you can specify host variable arrays to store the retrieved column values.

You can use host variable arrays to specify a program data area to contain multiple rows of column values. A DB2 *rowset cursor* enables an application to retrieve and process a set of rows from the result table of the cursor.

Related concepts:

“Host variable arrays in an SQL statement” on page 191

“Host variable arrays” on page 175

Related tasks:

“Accessing data by using a rowset-positioned cursor” on page 731

“Inserting multiple rows of data from host variable arrays”

Inserting multiple rows of data from host variable arrays

Use host variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

You can use a form of the INSERT statement or MERGE statement to insert multiple rows from values that are provided in host variable arrays. Each array contains values for a column of the target table. The first value in an array corresponds to the value for that column for the first inserted row, the second value in the array corresponds to the value for the column in the second inserted row, and so on. DB2 determines the attributes of the values based on the declaration of the array.

You can insert the number of rows that are specified in the host variable NUM-ROWS by using the following INSERT statement:

```

EXEC SQL
  INSERT INTO DSN8A10.ACT
    (ACTNO, ACTKWD, ACTDESC)
  VALUES (:HVA1, :HVA2, :HVA3)
  FOR :NUM-ROWS ROWS
END-EXEC.

```

Assume that the host variable arrays HVA1, HVA2, and HVA3 have been declared and populated with the values that are to be inserted into the ACTNO, ACTKWD, and ACTDESC columns. The NUM-ROWS host variable specifies the number of rows that are to be inserted, which must be less than or equal to the dimension of each host variable array.

Related tasks:

“Retrieving multiple rows of data into host variable arrays” on page 192

Retrieving a single row of data into a host structure

If you know that your query returns multiple column values for only one row, you can specify a host structure to contain the column values.

In the following example, assume that your COBOL program includes the following SQL statement:

```

EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT
  INTO :EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, :WORKDEPT
  FROM DSN8A10.VEMP
  WHERE EMPNO = :EMPID
END-EXEC.

```

If you want to avoid listing host variables, you can substitute the name of a structure, say :PEMP, that contains :EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, and :WORKDEPT. The example then reads:

```

EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT
  INTO :PEMP
  FROM DSN8A10.VEMP
  WHERE EMPNO = :EMPID
END-EXEC.

```

You can declare a host structure yourself, or you can use DCLGEN to generate a COBOL record description, PL/I structure declaration, or C structure declaration that corresponds to the columns of a table.

Related concepts:

“DCLGEN (declarations generator)” on page 161

“Host structures” on page 175

“Example: Adding DCLGEN declarations to a library” on page 169

Including dynamic SQL in your program

Dynamic SQL is prepared and executed while the program is running.

Before you use dynamic SQL, consider whether static SQL or dynamic SQL is the best technique for your application, and consider the type of dynamic SQL that you want to use. Also consider the performance implications of using dynamic SQL in application programs. For information about methods that you can use to improve the performance of dynamic SQL statements, see Improving dynamic SQL performance (DB2 Performance).

Introductory concepts:

Dynamic SQL applications (Introduction to DB2 for z/OS)

Dynamic SQL prepares and executes the SQL statements within a program, while the program is running.

You can issue dynamic SQL statements in the following contexts:

Interactive SQL

A user enters SQL statements through SPUIFI, the command line processor, or an interactive tool, such as DB2 QMF™ for Windows. DB2 prepares and executes those statements as dynamic SQL statements.

Embedded dynamic SQL

Your application puts the SQL source in host variables and includes PREPARE and EXECUTE statements that tell DB2 to prepare and run the contents of those host variables at run time. You must precompile and bind programs that include embedded dynamic SQL.

Deferred embedded SQL

Deferred embedded SQL statements are neither fully static nor fully dynamic. Like static statements, deferred embedded SQL statements are embedded within applications; however, like dynamic statements, they are prepared at run time. DB2 processes the deferred embedded SQL statements with bind-time rules. For example, DB2 uses the authorization ID and qualifier (that are determined at bind time) as the plan or package owner.

Dynamic SQL executed through ODBC or JDBC functions

Your application contains ODBC function calls that pass dynamic SQL statements as arguments. You do not need to precompile and bind programs that use ODBC function calls.

JDBC application support lets you write dynamic SQL applications in Java.

For most DB2 users, *static SQL*, which is embedded in a host language program and bound before the program runs, provides a straightforward, efficient path to DB2 data. You can use static SQL when you know before run time what SQL statements your application needs to execute.

Related tasks:

 Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Differences between static and dynamic SQL

Static and dynamic SQL are each appropriate for different circumstances. You should consider the differences between the two when determining whether static SQL or dynamic SQL is best for your application.

Flexibility of static SQL with host variables

Introductory concepts:

Static SQL (DB2 SQL)

Static SQL applications (Introduction to DB2 for z/OS)

Dynamic SQL applications (Introduction to DB2 for z/OS)

When you use static SQL, you cannot change the form of SQL statements unless you make changes to the program. However, you can increase the flexibility of static statements by using host variables.

Example: In the following example, the UPDATE statement can update the salary of any employee. At bind time, you know that salaries must be updated, but you do not know until run time whose salaries should be updated, and by how much.

```
01 IOAREA.  
   02 EMPID           PIC X(06).  
   02 NEW-SALARY      PIC S9(7)V9(2) COMP-3.  
   :  
   (Other declarations)  
READ CARDIN RECORD INTO IOAREA  
  AT END MOVE 'N' TO INPUT-SWITCH.  
  :  
  (Other COBOL statements)  
EXEC SQL  
  UPDATE DSN8A10.EMP  
    SET SALARY = :NEW-SALARY  
    WHERE EMPNO = :EMPID  
END-EXEC.
```

The statement (UPDATE) does not change, nor does its basic structure, but the input can change the results of the UPDATE statement.

Flexibility of dynamic SQL

What if a program must use different types and structures of SQL statements? If there are so many types and structures that it cannot contain a model of each one, your program might need dynamic SQL.

You can use one of the following programs to execute dynamic SQL:

DB2 Query Management Facility™ (DB2 QMF)

Provides an alternative interface to DB2 that accepts almost any SQL statement

SPUFI

Accepts SQL statements from an input data set, and then processes and executes them dynamically

command line processor

Accepts SQL statements from a UNIX System Services environment.

Limitations of dynamic SQL

You cannot use some of the SQL statements dynamically.

| For reactive governing cases, the ASUTIME limit specified for the top-level calling
| package is applied for the entire thread, regardless of any value specified for the
| routines that are called.

Dynamic SQL processing

A program that provides for dynamic SQL accepts as input, or generates, an SQL statement in the form of a character string. You can simplify the programming if you can plan the program not to use SELECT statements, or to use only those that

return a known number of values of known types. In the most general case, in which you do not know in advance about the SQL statements that will execute, the program typically takes these steps:

1. Translates the input data, including any parameter markers, into an SQL statement
2. Prepares the SQL statement to execute and acquires a description of the result table
3. Obtains, for SELECT statements, enough main storage to contain retrieved data
4. Executes the statement or fetches the rows of data
5. Processes the information returned
6. Handles SQL return codes.

Performance of static and dynamic SQL

To access DB2 data, an SQL statement requires an access path. Two big factors in the performance of an SQL statement are the amount of time that DB2 uses to determine the access path at run time and whether the access path is efficient. DB2 determines the access path for a statement at either of these times:

- When you bind the plan or package that contains the SQL statement
- When the SQL statement executes

The time at which DB2 determines the access path depends on these factors:

- Whether the statement is executed statically or dynamically
- Whether the statement contains input host variables
- Whether the statement contains a declared global temporary table.

Static SQL statements with no input host variables

For static SQL statements that do not contain input host variables, DB2 determines the access path when you bind the plan or package. This combination yields the best performance because the access path is already determined when the program executes.

Static SQL statements with input host variables

For static SQL statements that have input host variables, the time at which DB2 determines the access path depends on the REOPT bind option that you specify: REOPT(NONE) or REOPT(ALWAYS). REOPT(NONE) is the default. Do not specify REOPT(AUTO) or REOPT(ONCE); these options are applicable only to dynamic statements. DB2 ignores REOPT(ONCE) and REOPT(AUTO) for static SQL statements, because DB2 caches only dynamic SQL statements.

If you specify REOPT(NONE), DB2 determines the access path at bind time, just as it does when there are no input variables.

If you specify REOPT(ALWAYS), DB2 determines the access path at bind time and again at run time, using the values of the following types of input variables:

- Host variables
- Parameter markers
- Special registers

DB2 must spend extra time determining the access path for statements at run time. However if DB2 determines a significantly better access path using the variable values, you might see an overall performance improvement. With

REOPT(ALWAYS), DB2 optimizes statements using known literal values. Knowing the literal values can help DB2 to choose a more efficient access path when the columns contain skewed data. DB2 can also recognize which partitions qualify if there are search conditions with host variables on the limit keys of partitioned table spaces.

With REOPT(ALWAYS) DB2 does not start the optimization over from the beginning. For example DB2 does not perform query transformations based on the literal values. Consequently, static SQL statements that use host variables optimized with REOPT(ALWAYS) and similar SQL statements that use explicit literal values might result in different access paths.

Dynamic SQL statements

For dynamic SQL statements, DB2 determines the access path at run time, when the statement is prepared. The repeating cost of preparing a dynamic statement can make the performance worse than that of static SQL statements. However, if you execute the same SQL statement often, you can use the dynamic statement cache to decrease the number of times that those dynamic statements must be prepared.

Dynamic SQL statements with input host variables

When you bind applications that contain dynamic SQL statements with input host variables, consider using the REOPT(ALWAYS), REOPT(ONCE), or REOPT(AUTO) bind options, instead of the REOPT(NONE) option.

Use REOPT(ALWAYS) when you are not using the dynamic statement cache. DB2 determines the access path for statements at each EXECUTE or OPEN of the statement. This option ensures the best access path for a statement, but using REOPT(ALWAYS) can increase the cost of frequently used dynamic SQL statements.

Consequently, the REOPT(ALWAYS) option is not a good choice for high-volume sub-second queries. For high-volume fast running queries, the repeating cost of prepare can exceed the execution cost of the statement. Statements that are processed under the REOPT(ALWAYS) option are excluded from the dynamic statement cache even if dynamic statement caching is enabled because DB2 cannot reuse access paths when REOPT(ALWAYS) is specified.

Use REOPT(ONCE) or REOPT(AUTO) when you are using the dynamic statements cache:

- If you specify REOPT(ONCE), DB2 determines and the access path for statements only at the first EXECUTE or OPEN of the statement. It saves that access path in the dynamic statement cache and uses it until the statement is invalidated or removed from the cache. This reuse of the access path reduces the prepare cost of frequently used dynamic SQL statements that contain input host variables; however, it does not account for changes to parameter marker values for dynamic statements.

The REOPT(ONCE) option is ideal for ad-hoc query applications such as SPUIF, DSNTEP2, DSNTEP4, DSNTIAUL, and QMF. DB2 can better optimize statements knowing the literal values for special registers such as CURRENT DATE and CURRENT TIMESTAMP, rather than using default filter factor estimates.

- If you specify REOPT(AUTO), DB2 determines the access path at run time. For each execution of a statement with parameter markers, DB2 generates a new access path if it determines that a new access path is likely to improve performance.

Coding PREPARE statements for efficient optimization

You should code your PREPARE statements to minimize overhead. With REOPT(AUTO), REOPT(ALWAYS), and REOPT(ONCE), DB2 prepares an SQL statement at the same time as it processes OPEN or EXECUTE for the statement. That is, DB2 processes the statement as if you specify DEFER(PREPARE). However, in the following cases, DB2 prepares the statement twice:

- If you execute the DESCRIBE statement before the PREPARE statement in your program
- If you use the PREPARE statement with the INTO parameter

For the first prepare, DB2 determines the access path without using input variable values. For the second prepare, DB2 uses the input variable values to determine the access path. This extra prepare can decrease performance.

If you specify REOPT(ALWAYS), DB2 prepares the statement twice each time it is run.

If you specify REOPT(ONCE), DB2 prepares the statement twice only when the statement has never been saved in the cache. If the statement has been prepared and saved in the cache, DB2 will use the saved version of the statement to complete the DESCRIBE statement.

If you specify REOPT(AUTO), DB2 initially prepares the statement without using input variable values. If the statement has been saved in the cache, for the subsequent OPEN or EXECUTE, DB2 determines if a new access path is needed according to the input variable values.

For a statement that uses a cursor, you can avoid the double prepare by placing the DESCRIBE statement after the OPEN statement in your program.

If you use predictive governing, and a dynamic SQL statement that is bound with either REOPT(ALWAYS) or REOPT(ONCE) exceeds a predictive governing warning threshold, your application does not receive a warning SQLCODE. However, it will receive an error SQLCODE from the OPEN or EXECUTE statement.

Related tasks:

 [Reoptimizing SQL statements at run time \(DB2 Performance\)](#)

Related reference:

 [Characteristics of SQL statements in DB2 for z/OS \(DB2 SQL\)](#)

 [REOPT bind option \(DB2 Commands\)](#)

Possible host languages for dynamic SQL applications

Programs that use dynamic SQL are usually written in assembler, C, PL/I, REXX, and COBOL. All SQL statements in REXX programs are considered dynamic SQL.

You can write non-SELECT and fixed-list SELECT statements in any of the DB2 supported languages. A program containing a varying-list SELECT statement is more difficult to write in Fortran, because the program cannot run without the help of a subroutine to manage address variables (pointers) and storage allocation.

Most of the examples in this topic are in PL/I. Longer examples in the form of complete programs are available in the sample applications:

DSNTEP2

Processes both SELECT and non-SELECT statements dynamically. (PL/I).

DSNTIAD

Processes only non-SELECT statements dynamically. (Assembler).

DSNTIAUL

Processes SELECT statements dynamically. (Assembler).

Library *prefix*.SDSNSAMP contains the sample programs. You can view the programs online, or you can print them using ISPF, IEBPTPCH, or your own printing program.

You can use all forms of dynamic SQL in all supported versions of COBOL.

Related concepts:

“Sample COBOL dynamic SQL program” on page 364

Including dynamic SQL for non-SELECT statements in your program

The easiest way to use dynamic SQL is to use non-SELECT statements. Because you do not need to dynamically allocate any main storage, you can write your program in any host language, including Fortran.

Your program must take the following steps:

1. Include an SQLCA. The requirements for an SQL communications area (SQLCA) are the same as for static SQL statements. For REXX, DB2 includes the SQLCA automatically.
2. Load the input SQL statement into a data area. The procedure for building or reading the input SQL statement is not discussed here; the statement depends on your environment and sources of information. You can read in complete SQL statements, or you can get information to build the statement from data sets, a user at a terminal, previously set program variables, or tables in the database. If you attempt to execute an SQL statement dynamically that DB2 does not allow, you get an SQL error.
3. Execute the statement. You can use either of these methods:
 - EXECUTE IMMEDIATE
 - PREPARE and EXECUTE
4. Handle any errors that might result. The requirements are the same as those for static SQL statements. The return code from the most recently executed SQL statement appears in the host variables SQLCODE and SQLSTATE or corresponding fields of the SQLCA.

Related concepts:

“Sample dynamic and static SQL in a C program” on page 311

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Related tasks:

“Checking the execution of SQL statements” on page 227

“Dynamically executing an SQL statement by using EXECUTE IMMEDIATE” on page 219

“Dynamically executing an SQL statement by using PREPARE and EXECUTE” on page 221

Including dynamic SQL for fixed-list SELECT statements in your program

A fixed-list SELECT statement returns rows that contain a known number of values of a known type. When you use this type of statement, you know in advance exactly what kinds of host variables you need to declare to store the results.

The term “fixed-list” does not imply that you must know in advance how many rows of data will be returned. However, you must know the number of columns and the data types of those columns. A fixed-list SELECT statement returns a result table that can contain any number of rows; your program looks at those rows one at a time, using the FETCH statement. Each successive fetch returns the same number of values as the last, and the values have the same data types each time. Therefore, you can specify host variables as you do for static SQL.

An advantage of the fixed-list SELECT is that you can write it in any of the programming languages that DB2 supports. Varying-list dynamic SELECT statements require assembler, C, PL/I, and COBOL.

To execute a fixed-list SELECT statement dynamically, your program must:

1. Include an SQLCA.
2. Load the input SQL statement into a data area. The preceding two steps are exactly the same including dynamic SQL for non-SELECT statements in your program.
3. Declare a cursor for the statement name.
4. Prepare the statement.
5. Open the cursor.
6. Fetch rows from the result table.
7. Close the cursor.
8. Handle any resulting errors. This step is the same as for static SQL, except for the number and types of errors that can result.

Example: Suppose that your program retrieves last names and phone numbers by dynamically executing SELECT statements of this form:

```
SELECT LASTNAME, PHONENO FROM DSN8A10.EMP
WHERE ... ;
```

The program reads the statements from a terminal, and the user determines the WHERE clause.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Eventually you prepare a statement from DSTRING, but first you must declare a cursor for the statement and give it a name.

Declaring a cursor for the statement name:

Dynamic SELECT statements cannot use INTO. Therefore, you must use a cursor to put the results into host variables.

Example: When you declare the cursor, use the statement name (call it STMT), and give the cursor itself a name (for example, C1):

```
EXEC SQL DECLARE C1 CURSOR FOR STMT;
```

Preparing the statement:

Prepare a statement (STMT) from DSTRING.

Example: This is one possible PREPARE statement:

```
EXEC SQL PREPARE STMT FROM :DSTRING ATTRIBUTES :ATTRVAR;
```

ATTRVAR contains attributes that you want to add to the SELECT statement, such as FETCH FIRST 10 ROWS ONLY or OPTIMIZE for 1 ROW. In general, if the SELECT statement has attributes that conflict with the attributes in the PREPARE statement, the attributes on the SELECT statement take precedence over the attributes on the PREPARE statement. However, in this example, the SELECT statement in DSTRING has no attributes specified, so DB2 uses the attributes in ATTRVAR for the SELECT statement.

As with non-SELECT statements, the fixed-list SELECT could contain parameter markers. However, this example does not need them.

To execute STMT, your program must open the cursor, fetch rows from the result table, and close the cursor.

Opening the cursor:

The OPEN statement evaluates the SELECT statement named STMT.

Example: Without parameter markers, use this statement:

```
EXEC SQL OPEN C1;
```

If STMT contains parameter markers, you must use the USING clause of OPEN to provide values for all of the parameter markers in STMT.

Example: If four parameter markers are in STMT, you need the following statement:

```
EXEC SQL OPEN C1 USING :PARM1, :PARM2, :PARM3, :PARM4;
```

Fetching rows from the result table:

Example: Your program could repeatedly execute a statement such as this:

```
EXEC SQL FETCH C1 INTO :NAME, :PHONE;
```

The key feature of this statement is the use of a list of host variables to receive the values returned by FETCH. The list has a known number of items (in this case, two items, :NAME and :PHONE) of known data types (both are character strings, of lengths 15 and 4, respectively).

You can use this list in the FETCH statement only because you planned the program to use only fixed-list SELECTs. Every row that cursor C1 points to must contain exactly two character values of appropriate length. If the program is to handle anything else, it must use the techniques for including dynamic SQL for varying-list SELECT statements in your program.

Closing the cursor:

This step is the same as for static SQL.

Example: A WHENEVER NOT FOUND statement in your program can name a routine that contains this statement:

```
EXEC SQL CLOSE C1;
```

Related concepts:

“Sample dynamic and static SQL in a C program” on page 311

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Related tasks:

“Including dynamic SQL for non-SELECT statements in your program” on page 199

“Including dynamic SQL for varying-list SELECT statements in your program”

Including dynamic SQL for varying-list SELECT statements in your program

A varying-list SELECT statement returns rows that contain an unknown number of values of unknown type. When you use this type of statement, you do not know in advance exactly what kinds of host variables you need to declare for storing the results.

Because the varying-list SELECT statement requires pointer variables for the SQL descriptor area, you cannot issue it from a Fortran program. A Fortran program can call a subroutine written in a language that supports pointer variables (such as PL/I or assembler), if you need to use a varying-list SELECT statement.

What your application program must do for varying-list SELECT statements: To execute a varying-list SELECT statement dynamically, your program must follow these steps:

1. Include an SQLCA.
DB2 performs this step for a REXX program.
2. Load the input SQL statement into a data area.
3. Prepare and execute the statement. This step is more complex than for fixed-list SELECTs. It involves the following steps:
 - a. Include an SQLDA (SQL descriptor area).
DB2 performs this step for a REXX program.
 - b. Declare a cursor and prepare the variable statement.
 - c. Obtain information about the data type of each column of the result table.
 - d. Determine the main storage needed to hold a row of retrieved data.
You do not perform this step for a REXX program.
 - e. Put storage addresses in the SQLDA to tell where to put each item of retrieved data.
 - f. Open the cursor.

- g. Fetch a row.
- h. Eventually close the cursor and free main storage.

Additional complications exist for statements with parameter markers.

- 4. Handle any errors that might result.

Preparing a varying-list SELECT statement:

Suppose that your program dynamically executes SQL statements, but this time without any limits on their form. Your program reads the statements from a terminal, and you know nothing about them in advance. They might not even be SELECT statements.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Your program goes on to prepare a statement from the variable and then give the statement a name; call it STMT.

Now, the program must find out whether the statement is a SELECT. If it is, the program must also find out how many values are in each row, and what their data types are. The information comes from an SQL descriptor area (SQLDA).

An SQL descriptor area:

The SQLDA is a structure that is used to communicate with your program, and storage for it is usually allocated dynamically at run time.

To include the SQLDA in a PL/I or C program, use:

```
EXEC SQL INCLUDE SQLDA;
```

For assembler, use this in the storage definition area of a CSECT:

```
EXEC SQL INCLUDE SQLDA
```

For COBOL, use:

```
EXEC SQL INCLUDE SQLDA END-EXEC.
```

You cannot include an SQLDA in a Fortran, or REXX program.

Obtaining information about the SQL statement:

An SQLDA can contain a variable number of occurrences of SQLVAR, each of which is a set of five fields that describe one column in the result table of a SELECT statement.

The number of occurrences of SQLVAR depends on the following factors:

- The number of columns in the result table you want to describe.
- Whether you want the PREPARE or DESCRIBE to put both column names and labels in your SQLDA. This is the option USING BOTH in the PREPARE or DESCRIBE statement.
- Whether any columns in the result table are LOB types or distinct types.

The following table shows the minimum number of SQLVAR instances you need for a result table that contains n columns.

Table 43. Minimum number of SQLVARs for a result table with n columns

Type of DESCRIBE and contents of result table	Not USING BOTH	USING BOTH
No distinct types or LOBs	n	$2*n$
Distinct types but no LOBs	$2*n$	$3*n$
LOBs but no distinct types	$2*n$	$2*n$
LOBs and distinct types	$2*n$	$3*n$

An SQLDA with n occurrences of SQLVAR is referred to as a *single SQLDA*, an SQLDA with $2*n$ occurrences of SQLVAR a *double SQLDA*, an SQLDA with $3*n$ occurrences of SQLVAR a *triple SQLDA*.

A program that admits SQL statements of every kind for dynamic execution has two choices:

- Provide the largest SQLDA that it could ever need. The maximum number of columns in a result table is 750, so an SQLDA for 750 columns occupies 33 016 bytes for a single SQLDA, 66 016 bytes for a double SQLDA, or 99 016 bytes for a triple SQLDA. Most SELECT statements do not retrieve 750 columns, so the program does not usually use most of that space.
- Provide a smaller SQLDA, with fewer occurrences of SQLVAR. From this the program can find out whether the statement was a SELECT and, if it was, how many columns are in its result table. If more columns are in the result than the SQLDA can hold, DB2 returns no descriptions. When this happens, the program must acquire storage for a second SQLDA that is long enough to hold the column descriptions, and ask DB2 for the descriptions again. Although this technique is more complicated to program than the first, it is more general.

How many columns should you allow? You must choose a number that is large enough for most of your SELECT statements, but not too wasteful of space; 40 is a good compromise. To illustrate what you must do for statements that return more columns than allowed, the example in this discussion uses an SQLDA that is allocated for at least 100 columns.

Declaring a cursor for the statement:

As before, you need a cursor for the dynamic SELECT. For example, write:

```
EXEC SQL
  DECLARE C1 CURSOR FOR STMT;
```

Preparing the statement using the minimum SQLDA:

Suppose that your program declares an SQLDA structure with the name MINSQLDA, having 100 occurrences of SQLVAR and SQLN set to 100. To prepare a statement from the character string in DSTRING and also enter its description into MINSQLDA, write this:

```
EXEC SQL PREPARE STMT FROM :DSTRING;
EXEC SQL DESCRIBE STMT INTO :MINSQLDA;
```

Equivalently, you can use the INTO clause in the PREPARE statement:

```
EXEC SQL
  PREPARE STMT INTO :MINSQLDA FROM :DSTRING;
```

Do not use the USING clause in either of these examples. At the moment, only the minimum SQLDA is in use. The following figure shows the contents of the

minimum SQLDA in use.



Figure 12. The minimum SQLDA structure

SQLN determines what SQLVAR gets:

The SQLN field, which you must set before using DESCRIBE (or PREPARE INTO), tells how many occurrences of SQLVAR the SQLDA is allocated for. If DESCRIBE needs more than that, the results of the DESCRIBE depend on the contents of the result table. Let n indicate the number of columns in the result table. Then:

- If the result table contains at least one distinct type column but no LOB columns, you do not specify USING BOTH, and $n \leq \text{SQLN} < 2 * n$, then DB2 returns base SQLVAR information in the first n SQLVAR occurrences, but no distinct type information. Base SQLVAR information includes:
 - Data type code
 - Length attribute (except for LOBs)
 - Column name or label
 - Host variable address
 - Indicator variable address
- Otherwise, if SQLN is less than the minimum number of SQLVARs specified in the table above, then DB2 returns no information in the SQLVARs.

Regardless of whether your SQLDA is big enough, whenever you execute DESCRIBE, DB2 returns the following values, which you can use to build an SQLDA of the correct size:

- SQLD is 0 if the SQL statement is not a SELECT. Otherwise, SQLD is the number of columns in the result table. The number of SQLVAR occurrences you need for the SELECT depends on the value in the seventh byte of SQLDAID.
- The seventh byte of SQLDAID is 2 if each column in the result table requires two SQLVAR entries. The seventh byte of SQLDAID is 3 if each column in the result table requires three SQLVAR entries.

If the statement is not a SELECT:

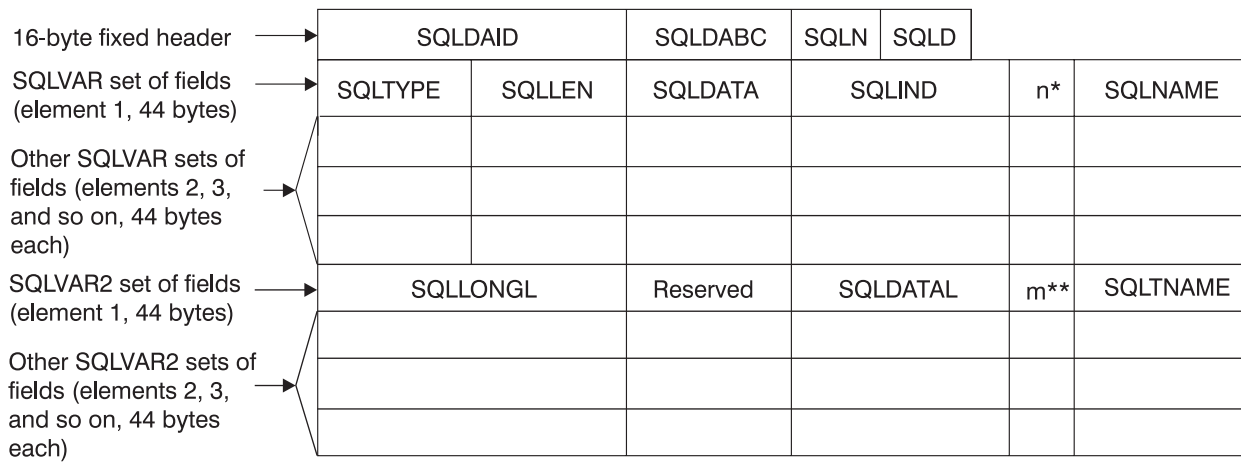
To find out if the statement is a SELECT, your program can query the SQLD field in MINSQLDA. If the field contains 0, the statement is not a SELECT, the statement is already prepared, and your program can execute it. If no parameter markers are in the statement, you can use:

```
EXEC SQL EXECUTE STMT;
```

(If the statement does contain parameter markers, you must use an SQL descriptor area)

Acquiring storage for a second SQLDA if needed:

Now you can allocate storage for a second, full-size SQLDA; call it FULSQLDA. The following figure shows its structure.



- * The length of the character string in SQLNAME. SQLNAME is a 30-byte area immediately following the length field.
- ** The length of the character string in SQLTNAME. SQLTNAME is a 30-byte area immediately following the length field.

Figure 13. The full-size SQLDA structure

FULSQLDA has a fixed-length header of 16 bytes in length, followed by a varying-length section that consists of structures with the SQLVAR format. If the result table contains LOB columns or distinct type columns, a varying-length section that consists of structures with the SQLVAR2 format follows the structures with SQLVAR format. All SQLVAR structures and SQLVAR2 structures are 44 bytes long. The number of SQLVAR and SQLVAR2 elements you need is in the SQLD field of MINSQLDA, and the total length you need for FULSQLDA (16 + SQLD * 44) is in the SQLDABC field of MINSQLDA. Allocate that amount of storage.

Describing the SELECT statement again:

After allocating sufficient space for FULSQLDA, your program must take these steps:

1. Put the total number of SQLVAR and SQLVAR2 occurrences in FULSQLDA into the SQLN field of FULSQLDA. This number appears in the SQLD field of MINSQLDA.
2. Describe the statement again into the new SQLDA:

```
EXEC SQL DESCRIBE STMT INTO :FULSQLDA;
```

After the DESCRIBE statement executes, each occurrence of SQLVAR in the full-size SQLDA (FULSQLDA in our example) contains a description of one column of the result table in five fields. If an SQLVAR occurrence describes a LOB column or distinct type column, the corresponding SQLVAR2 occurrence contains additional information specific to the LOB or distinct type.

The following figure shows an SQLDA that describes two columns that are not LOB columns or distinct type columns.

SQLDA header	SQLDA			8816	200	200
SQLVAR element 1 (44 bytes)	452	3	Undefined	0	8	WORKDEPT
SQLVAR element 2 (44 bytes)	453	4	Undefined	0	7	PHONENO

Figure 14. Contents of FULSQLDA after executing DESCRIBE

Acquiring storage to hold a row:

Before fetching rows of the result table, your program must:

1. Analyze each SQLVAR description to determine how much space you need for the column value.
2. Derive the address of some storage area of the required size.
3. Put this address in the SQLDATA field.

If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field. The following figures show the SQL descriptor area after you take certain actions.

In the previous figure, the DESCRIBE statement inserted all the values except the first occurrence of the number 200. The program inserted the number 200 before it executed DESCRIBE to tell how many occurrences of SQLVAR to allow. If the result table of the SELECT has more columns than this, the SQLVAR fields describe nothing.

The first SQLVAR pertains to the first column of the result table (the WORKDEPT column). SQLVAR element 1 contains fixed-length character strings and does not allow null values (SQLTYPE=452); the length attribute is 3.

The following figure shows the SQLDA after your program acquires storage for the column values and their indicators, and puts the addresses in the SQLDATA fields of the SQLDA.

SQLDA header	SQLDA			8816	200	200
SQLVAR element 1 (44 bytes)	452	3	Addr FLDA	Addr FLDAI	8	WORKDEPT
SQLVAR element 2 (44 bytes)	453	4	Addr FLDB	Addr FLDBI	7	PHONENO

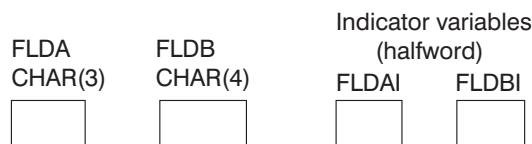


Figure 15. SQL descriptor area after analyzing descriptions and acquiring storage

The following figure shows the SQLDA after your program executes a FETCH statement.

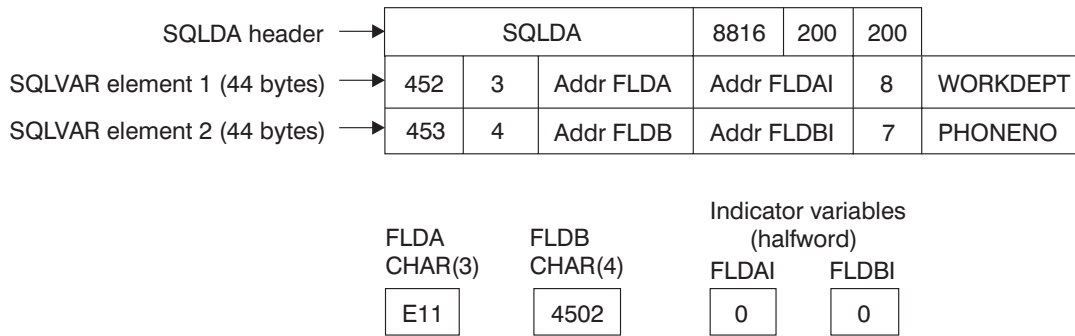


Figure 16. SQL descriptor area after executing *FETCH*

The following table describes the values in the descriptor area.

Table 44. Values inserted in the *SQLDA*

Value	Field	Description
SQLDA	SQLDAID	An "eye-catcher"
8816	SQLDABC	The size of the <i>SQLDA</i> in bytes (16 + 44 * 200)
200	SQLN	The number of occurrences of <i>SQLVAR</i> , set by the program
200	SQLD	The number of occurrences of <i>SQLVAR</i> actually used by the <i>DESCRIBE</i> statement
452	SQLTYPE	The value of <i>SQLTYPE</i> in the first occurrence of <i>SQLVAR</i> . It indicates that the first column contains fixed-length character strings, and does not allow nulls.
3	SQLLEN	The length attribute of the column
Undefined or CCSID value	SQLDATA	Bytes 3 and 4 contain the <i>CCSID</i> of a string column. Undefined for other types of columns.
Undefined	SQLIND	
8	SQLNAME	The number of characters in the column name
WORKDEPT	SQLNAME+2	The column name of the first column

Putting storage addresses in the *SQLDA*:

After analyzing the description of each column, your program must replace the content of each *SQLDATA* field with the address of a storage area large enough to hold values from that column. Similarly, for every column that allows nulls, the program must replace the content of the *SQLIND* field. The content must be the address of a halfword that you can use as an indicator variable for the column. The program can acquire storage for this purpose, of course, but the storage areas used do not have to be contiguous.

Figure 15 on page 207 shows the content of the descriptor area before the program obtains any rows of the result table. Addresses of fields and indicator variables are already in the *SQLVAR*.

Changing the *CCSID* for retrieved data:

All DB2 string data has an encoding scheme and *CCSID* associated with it. When you select string data from a table, the selected data generally has the same

encoding scheme and CCSID as the table. If the application uses some method, such as issuing the DECLARE VARIABLE statement, to change the CCSID of the selected data, the data is converted from the CCSID of the table to the CCSID that is specified by the application.

You can set the default application encoding scheme for a plan or package by specifying the value in the APPLICATION ENCODING field of the panel DEFAULTS FOR BIND PACKAGE or DEFAULTS FOR BIND PLAN. The default application encoding scheme for the DB2 subsystem is the value that was specified in the APPLICATION ENCODING field of installation panel DSNTIPF.

If you want to retrieve the data in an encoding scheme and CCSID other than the default values, you can use one of the following techniques:

- For dynamic SQL, set the CURRENT APPLICATION ENCODING SCHEME special register before you execute the SELECT statements. For example, to set the CCSID and encoding scheme for retrieved data to the default CCSID for Unicode, execute this SQL statement:

```
EXEC SQL SET CURRENT APPLICATION ENCODING SCHEME = 'UNICODE';
```

The initial value of this special register is the application encoding scheme that is determined by the BIND option.

- For static and dynamic SQL statements that use host variables and host variable arrays, use the DECLARE VARIABLE statement to associate CCSIDs with the host variables into which you retrieve the data. See “Setting the CCSID for host variables” on page 178 for information about this technique.
- For static and dynamic SQL statements that use a descriptor, set the CCSID for the retrieved data in the SQLDA. The following text describes that technique.

To change the encoding scheme for SQL statements that use a descriptor, set up the SQLDA, and then make these additional changes to the SQLDA:

1. Put the character + in the sixth byte of field SQLDAID.
2. For each SQLVAR entry:
 - a. Set the length field of SQLNAME to 8.
 - b. Set the first two bytes of the data field of SQLNAME to X'0000'.
 - c. Set the third and fourth bytes of the data field of SQLNAME to the CCSID, in hexadecimal, in which you want the results to display, or to X'0000'. X'0000' indicates that DB2 should use the default CCSID. If you specify a nonzero CCSID, it must meet one of the following conditions:
 - A row in catalog table SYSSTRINGS has a matching value for OUTCCSID.
 - The Unicode conversion services support conversion to that CCSID. See z/OS C/C++ Programming Guide for information about the conversions supported.

If you are modifying the CCSID to retrieve the contents of an ASCII, EBCDIC, or Unicode table on a DB2 for z/OS system, and you previously executed a DESCRIBE statement on the SELECT statement that you are using to retrieve the data, the SQLDATA fields in the SQLDA that you used for the DESCRIBE contain the ASCII or Unicode CCSID for that table. To set the data portion of the SQLNAME fields for the SELECT, move the contents of each SQLDATA field in the SQLDA from the DESCRIBE to each SQLNAME field in the SQLDA for the SELECT. If you are using the same

SQLDA for the DESCRIBE and the SELECT, be sure to move the contents of the SQLDATA field to SQLNAME before you modify the SQLDATA field for the SELECT.

For REXX, you set the CCSID in the *stem.n*.SQLUSECCSID field instead of setting the SQLDAID and SQLNAME fields.

For example, suppose that the table that contains WORKDEPT and PHONENO is defined with CCSID ASCII. To retrieve data for columns WORKDEPT and PHONENO in ASCII CCSID 437 (X'01B5'), change the SQLDA as shown in the following figure.

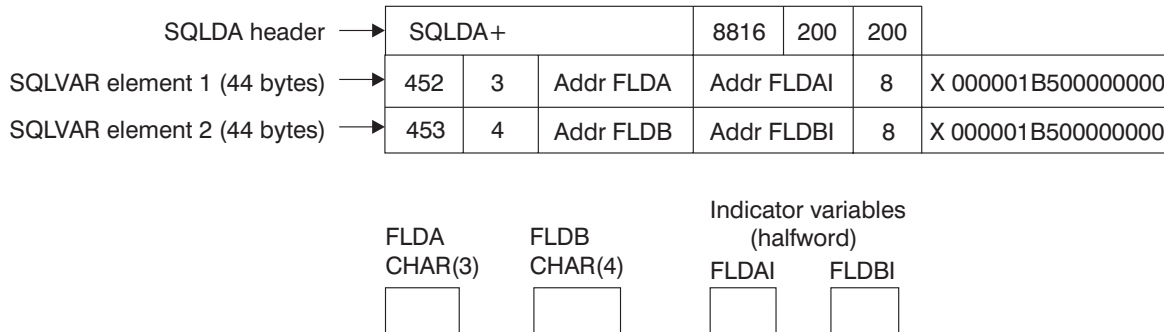


Figure 17. SQL descriptor area for retrieving data in ASCII CCSID 437

Specifying that DESCRIBE use column labels in the SQLNAME field:

By default, DESCRIBE describes each column in the SQLNAME field by the column name. You can tell it to use column labels instead.

Restriction: You cannot use column labels with set operators (UNION, INTERSECT, and EXCEPT).

To specify that DESCRIBE use column labels in the SQLNAME field, specify one of the following options when you issue the DESCRIBE statement:

USING LABELS

Specifies that SQLNAME is to contain labels. If a column has no label, SQLNAME contains nothing.

USING ANY

Specifies that SQLNAME is to contain labels wherever they exist. If a column has no label, SQLNAME contains the column name.

USING BOTH

Specifies that SQLNAME is to contain both labels and column names, when both exist.

In this case, FULSQLDA must contain a second set of occurrences of SQLVAR. The first set contains descriptions of all the columns with column names; the second set contains descriptions with column labels.

If you choose this option, perform the following actions:

- Allocate a longer SQLDA for the second DESCRIBE statement ((16 + SQLD * 88 bytes) instead of (16 + SQLD * 44))
- Put double the number of columns (SQLD * 2) in the SQLN field of the second SQLDA.

These actions ensure that enough space is available. Otherwise, if not enough space is available, DESCRIBE does not enter descriptions of any of the columns.

```
EXEC SQL
  DESCRIBE STMT INTO :FULSQLDA USING LABELS;
```

Some columns, such as those derived from functions or expressions, have neither name nor label; SQLNAME contains nothing for those columns. For example, if you use a UNION to combine two columns that do not have the same name and do not use a label, SQLNAME contains a string of length zero.

Describing tables with LOB and distinct type columns:

In general, the steps that you perform when you prepare an SQLDA to select rows from a table with LOB and distinct type columns are similar to the steps that you perform if the table has no columns of this type. The only difference is that you need to analyze some additional fields in the SQLDA for LOB or distinct type columns.

Example: Suppose that you want to execute this SELECT statement:

```
SELECT USER, A_DOC FROM DOCUMENTS;
```

The USER column cannot contain nulls and is of distinct type ID, defined like this:

```
CREATE DISTINCT TYPE SCHEMA1.ID AS CHAR(20);
```

The A_DOC column can contain nulls and is of type CLOB(1M).

The result table for this statement has two columns, but you need four SQLVAR occurrences in your SQLDA because the result table contains a LOB type and a distinct type. Suppose that you prepare and describe this statement into FULSQLDA, which is large enough to hold four SQLVAR occurrences. FULSQLDA looks like the following figure .

SQLDA header	SQLDA 2			192	4	4
SQLVAR element 1 (44 bytes)	452	20	Undefined	0	4	USER
SQLVAR element 2 (44 bytes)	409	0	Undefined	0	5	A_DOC
SQLVAR2 element 1 (44 bytes)					7	SCH1.ID
SQLVAR2 element 2 (44 bytes)	1 048 576				11	SYSIBM.CLOB

Figure 18. SQL descriptor area after describing a CLOB and distinct type

The next steps are the same as for result tables without LOBs or distinct types:

1. Analyze each SQLVAR description to determine the maximum amount of space you need for the column value.
For a LOB type, retrieve the length from the SQLLONGGL field instead of the SQLLEN field.
2. Derive the address of some storage area of the required size.
For a LOB data type, you also need a 4-byte storage area for the length of the LOB data. You can allocate this 4-byte area at the beginning of the LOB data or in a different location.
3. Put this address in the SQLDATA field.

For a LOB data type, if you allocated a separate area to hold the length of the LOB data, put the address of the length field in SQLDATAL. If the length field is at beginning of the LOB data area, put 0 in SQLDATAL. When you use a file reference variable for a LOB column, the indicator variable indicates whether the data in the file is null, not whether the data to which SQLDATA points is null.

4. If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field.

The following figure shows the contents of FULSQLDA after you enter pointers to the storage locations.

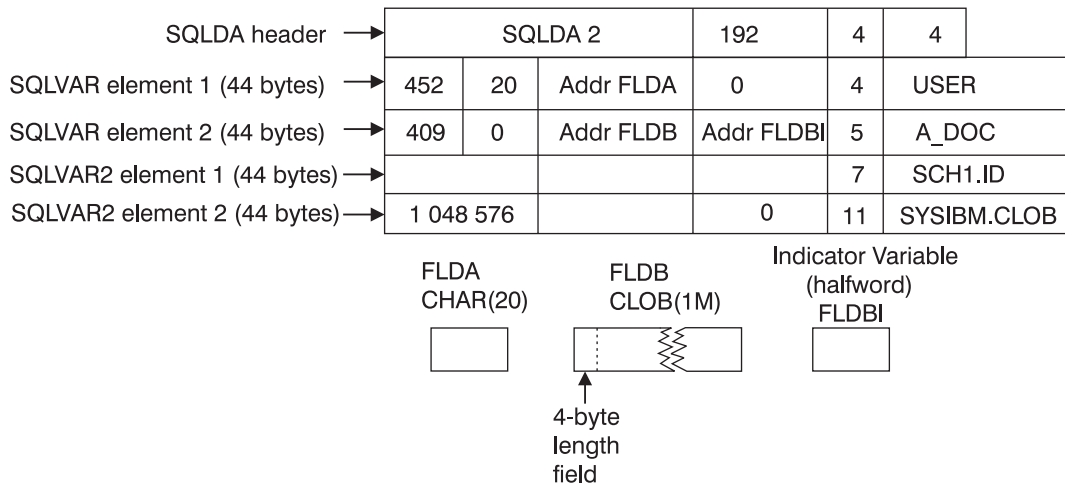


Figure 19. SQL descriptor area after analyzing CLOB and distinct type descriptions and acquiring storage

The following figure shows the contents of FULSQLDA after you execute a FETCH statement.

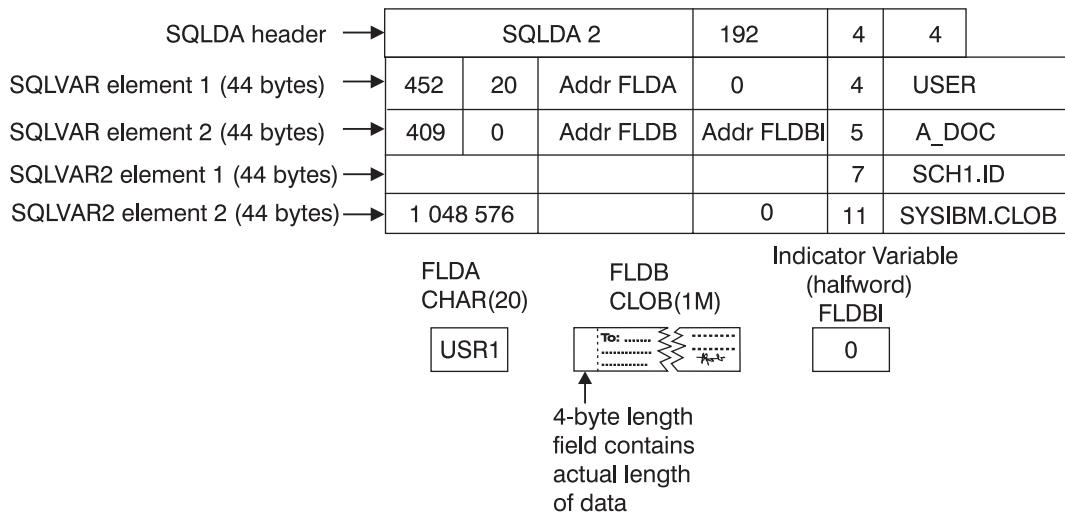


Figure 20. SQL descriptor area after executing FETCH on a table with CLOB and distinct type columns

Setting an XML host variable in an SQLDA:

Instead of specifying host variables to store XML values from a table, you can create an SQLDA to point to the data areas where DB2 puts the retrieved data. The SQLDA needs to describe the data type for each data area.

To set an XML host variable in an SQLDA:

1. Allocate an appropriate SQLDA.
2. Issue a DESCRIBE statement for the SQL statement whose result set you want to store. The DESCRIBE statement populates the SQLDA based on the column definitions. In the SQLDA, an SQLVAR entry is populated for each column in the result set. (Multiple SQLVAR entries are populated for LOB columns and columns with distinct types.) For columns of type XML the associated SQLVAR entry is populated as follows:

Table 45. SQLVAR field values for XML columns

SQLVAR field	Value for an XML column
sqltype SQLTYPE	988 for a column that is not nullable or 989 for a nullable column
sqlen SQLLEN	0
sqldata SQLDATA	0
sqlind SQLIND	0
sqlname SQLNAME	The unqualified name or label of the column

3. Check the SQLTYPE field of each SQLVAR entry. If the SQLTYPE field is 988 or 989, the column in the result set is an XML column.
4. For each XML column, make the following changes to the associated SQLVAR entry:
 - a. Change the SQLTYPE field to indicate the data type of the host variable to receive the XML data. You can retrieve the XML data into a host variable of type XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, or a compatible string data type.

If the target host variable type is XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, set the SQLTYPE field to one of the following values:

- 404**
XML AS BLOB
- 405**
nullable XML AS BLOB
- 408**
XML AS CLOB
- 409**
nullable XML AS CLOB
- 412**
XML AS DBCLOB

413

nullable XML AS DBCLOB

If the target host variable type is a string data type, set the SQLTYPE field to a valid string value.

Restriction: You cannot use the XML type (988/989) as a target host variable type.

- b. If the target host variable type is XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, change the first two bytes in the SQLNAME field to X'0000' and the fifth and sixth bytes to X'0100'. These bytes indicate that the value to be received is an XML value.
5. Populate the extended SQLVAR fields for each XML column as you would for a LOB column, as indicated in the following table.

Table 46. Fields for an extended SQLVAR entry for an XML host variable

SQLVAR field	Value for an XML host variable
len.sqllonglen SQLLONGL SQLLONGLEN	length attribute for the XML host variable
*	Reserved
sqldatalen SQLDATAL SQLDATALEN	pointer to the length of the XML host variable
sqldatatype_name SQLTNAME SQLDATATYPENAME	not used

You can now use the SQLDA to retrieve the XML data into a host variable of type XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, or a compatible string data type.

Executing a varying-list SELECT statement dynamically:

You can easily retrieve rows of the result table using a varying-list SELECT statement. The statements differ only a little from those for the fixed-list example.

Open the cursor: If the SELECT statement contains no parameter marker, this step is simple enough. For example:

```
EXEC SQL OPEN C1;
```

Fetch rows from the result table: This statement differs from the corresponding one for the case of a fixed-list select. Write:

```
EXEC SQL
  FETCH C1 USING DESCRIPTOR :FULSQLDA;
```

The key feature of this statement is the clause USING DESCRIPTOR :FULSQLDA. That clause names an SQL descriptor area in which the occurrences of SQLVAR point to other areas. Those other areas receive the values that FETCH returns. It is possible to use that clause only because you previously set up FULSQLDA to look like Figure 14 on page 207.

Figure 16 on page 208 shows the result of the FETCH. The data areas identified in the SQLVAR fields receive the values from a single row of the result table.

Successive executions of the same FETCH statement put values from successive rows of the result table into these same areas.

Close the cursor: This step is the same as for the fixed-list case. When no more rows need to be processed, execute the following statement:

```
EXEC SQL CLOSE C1;
```

When COMMIT ends the unit of work containing OPEN, the statement in STMT reverts to the unprepared state. Unless you defined the cursor using the WITH HOLD option, you must prepare the statement again before you can reopen the cursor.

Executing arbitrary statements with parameter markers:

Consider, as an example, a program that executes dynamic SQL statements of several kinds, including varying-list SELECT statements, any of which might contain a variable number of parameter markers. This program might present your users with lists of choices: choices of operation (update, select, delete); choices of table names; choices of columns to select or update. The program also enables the users to enter lists of employee numbers to apply to the chosen operation. From this, the program constructs SQL statements of several forms, one of which looks like this:

```
SELECT .... FROM DSN8A10.EMP  
WHERE EMPNO IN (?, ?, ?, ...?);
```

The program then executes these statements dynamically.

When the number and types of parameters are known: In the preceding example, you do not know in advance the number of parameter markers, and perhaps the kinds of parameter they represent. You can use techniques described previously if you know the number and types of parameters, as in the following examples:

- If the SQL statement is **not** SELECT, name a list of host variables in the EXECUTE statement:
WRONG: EXEC SQL EXECUTE STMT;
RIGHT: EXEC SQL EXECUTE STMT USING :VAR1, :VAR2, :VAR3;
- If the SQL statement is SELECT, name a list of host variables in the OPEN statement:
WRONG: EXEC SQL OPEN C1;
RIGHT: EXEC SQL OPEN C1 USING :VAR1, :VAR2, :VAR3;

In **both** cases, the number and types of host variables named must agree with the number of parameter markers in STMT and the types of parameter they represent. The first variable (VAR1 in the examples) must have the type expected for the first parameter marker in the statement, the second variable must have the type expected for the second marker, and so on. There must be at least as many variables as parameter markers.

When the number and types of parameters are not known: When you do not know the number and types of parameters, you can adapt the SQL descriptor area.

Your program can include an unlimited number of SQLDAs, and you can use them for different purposes. Suppose that an SQLDA, arbitrarily named DPARM, describes a set of parameters.

The structure of DPARM is the same as that of any other SQLDA. The number of occurrences of SQLVAR can vary, as in previous examples. In this case, every parameter marker must have one SQLVAR. Each occurrence of SQLVAR describes one host variable that replaces one parameter marker at run time. DB2 replaces the parameter markers when a non-SELECT statement executes or when a cursor is opened for a SELECT statement.

You must enter certain fields in DPARM **before** using EXECUTE or OPEN; you can ignore the other fields.

Field Use when describing host variables for parameter markers

SQLDAID

The seventh byte indicates whether more than one SQLVAR entry is used for each parameter marker. If this byte is not blank, at least one parameter marker represents a distinct type or LOB value, so the SQLDA has more than one set of SQLVAR entries.

You do not set this field for a REXX SQLDA.

SQLDABC

The length of the SQLDA, which is equal to $SQLN * 44 + 16$. You do not set this field for a REXX SQLDA.

SQLN The number of occurrences of SQLVAR allocated for DPARM. You do not set this field for a REXX SQLDA.

SQLD The number of occurrences of SQLVAR actually used. This number must not be less than the number of parameter markers. In each occurrence of SQLVAR, put information in the following fields: SQLTYPE, SQLLEN, SQLDATA, SQLIND.

SQLTYPE

The code for the type of variable, and whether it allows nulls.

SQLLEN

The length of the host variable.

SQLDATA

The address of the host variable.

For REXX, this field contains the value of the host variable.

SQLIND

The address of an indicator variable, if needed.

For REXX, this field contains a negative number if the value in SQLDATA is null.

SQLNAME

Ignore.

Using the SQLDA with EXECUTE or OPEN: To indicate that the SQLDA called DPARM describes the host variables substituted for the parameter markers at run time, use a USING DESCRIPTOR clause with EXECUTE or OPEN.

- For a non-SELECT statement, write:
EXEC SQL EXECUTE STMT USING DESCRIPTOR :DPARM;
- For a SELECT statement, write:

```
EXEC SQL OPEN C1 USING DESCRIPTOR :DPARM;
```

How bind options REOPT(ALWAYS), REOPT(AUTO) and REOPT(ONCE) affect dynamic SQL:

When you specify the bind option REOPT(ALWAYS), DB2 reoptimizes the access path at run time for SQL statements that contain host variables, parameter markers, or special registers. The option REOPT(ALWAYS) has the following effects on dynamic SQL statements:

- When you specify the option REOPT(ALWAYS), DB2 automatically uses DEFER(PREPARE), which means that DB2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.
- When you execute a DESCRIBE statement and then an EXECUTE statement on a non-SELECT statement, DB2 prepares the statement twice: Once for the DESCRIBE statement and once for the EXECUTE statement. DB2 uses the values in the input variables only during the second PREPARE. These multiple PREPAREs can cause performance to degrade if your program contains many dynamic non-SELECT statements. To improve performance, consider putting the code that contains those statements in a separate package and then binding that package with the option REOPT(NONE).
- If you execute a DESCRIBE statement before you open a cursor for that statement, DB2 prepares the statement twice. If, however, you execute a DESCRIBE statement after you open the cursor, DB2 prepares the statement only once. To improve the performance of a program bound with the option REOPT(ALWAYS), execute the DESCRIBE statement **after** you open the cursor. To prevent an automatic DESCRIBE before a cursor is opened, do not use a PREPARE statement with the INTO clause.
- If you use predictive governing for applications bound with REOPT(ALWAYS), DB2 does not return a warning SQLCODE when dynamic SQL statements exceed the predictive governing warning threshold. DB2 does return an error SQLCODE when dynamic SQL statements exceed the predictive governing error threshold. DB2 returns the error SQLCODE for an EXECUTE or OPEN statement.

When you specify the bind option REOPT(AUTO), DB2 optimizes the access path for SQL statements at the first EXECUTE or OPEN. Each time a statement is executed, DB2 determines if a new access path is needed to improve the performance of the statement. If a new access path will improve the performance, DB2 generates one. The option REOPT(AUTO) has the following effects on dynamic SQL statements:

- When you specify the bind option REOPT(AUTO), DB2 optimizes the access path for SQL statements at the first EXECUTE or OPEN. Each time a statement is executed, DB2 determines if a new access path is needed to improve the performance of the statement. If a new access path will improve the performance, DB2 generates one.
- When you specify the option REOPT(ONCE), DB2 automatically uses DEFER(PREPARE), which means that DB2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.
- When DB2 prepares a statement using REOPT(AUTO), it saves the access path in the dynamic statement cache. This access path is used each time the statement is run, until DB2 determines that a new access path is needed to improve the performance or the statement that is in the cache is invalidated (or removed from the cache) and needs to be rebound.

- The DESCRIBE statement has the following effects on dynamic statements that are bound with REOPT(AUTO):
 - When you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement, DB2 prepares the statement an extra time if it is not already saved in the cache: Once for the DESCRIBE statement and once for the EXECUTE statement. DB2 uses the values of the input variables only during the second time the statement is prepared. It then saves the statement in the cache. If you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement that has already been saved in the cache, DB2 will always prepare the non-SELECT statement for the DESCRIBE statement, and will prepare the statement again on EXECUTE only if DB2 determines that a new access path different from the one already saved in the cache can improve the performance.
 - If you execute DESCRIBE on a statement before you open a cursor for that statement, DB2 always prepares the statement on DESCRIBE. However, DB2 will not prepare the statement again on OPEN if the statement has already been saved in the cache and DB2 does not think that a new access path is needed at OPEN time. If you execute DESCRIBE on a statement after you open a cursor for that statement, DB2 prepared the statement only once if it is not already saved in the cache. If the statement is already saved in the cache and you execute DESCRIBE after you open a cursor for that statement, DB2 does not prepare the statement, it used the statement that is saved in the cache.
- If you use predictive governing for applications that are bound with REOPT(AUTO), DB2 does not return a warning SQLCODE when dynamic SQL statements exceed the predictive governing warning threshold. DB2 does return an error SQLCODE when dynamic SQL statements exceed the predictive governing error threshold. DB2 returns the error SQLCODE for an EXECUTE or OPEN statement.

When you specify the bind option REOPT(ONCE), DB2 optimizes the access path only once, at the first EXECUTE or OPEN, for SQL statements that contain host variables, parameter markers, or special registers. The option REOPT(ONCE) has the following effects on dynamic SQL statements:

- When you specify the option REOPT(ONCE), DB2 automatically uses DEFER(PREPARE), which means that DB2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.
- When DB2 prepares a statement using REOPT(ONCE), it saves the access path in the dynamic statement cache. This access path is used each time the statement is run, until the statement that is in the cache is invalidated (or removed from the cache) and needs to be rebound.
- The DESCRIBE statement has the following effects on dynamic statements that are bound with REOPT(ONCE):
 - When you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement, DB2 prepares the statement twice if it is not already saved in the cache: Once for the DESCRIBE statement and once for the EXECUTE statement. DB2 uses the values of the input variables only during the second time the statement is prepared. It then saves the statement in the cache. If you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement that has already been saved in the cache, DB2 prepares the non-SELECT statement only for the DESCRIBE statement.
 - If you execute DESCRIBE on a statement **before** you open a cursor for that statement, DB2 always prepares the statement on DESCRIBE. However, DB2 will not prepare the statement again on OPEN if the statement has already

been saved in the cache. If you execute DESCRIBE on a statement **after** you open a cursor for that statement, DB2 prepared the statement only once if it is not already saved in the cache. If the statement is already saved in the cache and you execute DESCRIBE after you open a cursor for that statement, DB2 does not prepare the statement, it used the statement that is saved in the cache.

To improve the performance of a program that is bound with REOPT(ONCE), execute the DESCRIBE statement after you open a cursor. To prevent an automatic DESCRIBE before a cursor is opened, do not use a PREPARE statement with the INTO clause.

- If you use predictive governing for applications that are bound with REOPT(ONCE), DB2 does not return a warning SQLCODE when dynamic SQL statements exceed the predictive governing warning threshold. DB2 does return an error SQLCODE when dynamic SQL statements exceed the predictive governing error threshold. DB2 returns the error SQLCODE for an EXECUTE or OPEN statement.

Related concepts:

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Related reference:

[➤ DESCRIBE OUTPUT \(DB2 SQL\)](#)

[➤ SQL descriptor area \(SQLDA\) \(DB2 SQL\)](#)

[➤ SQLTYPE and SQLLEN \(DB2 SQL\)](#)

[➤ The SQLDA Header \(DB2 SQL\)](#)

Dynamically executing an SQL statement by using EXECUTE IMMEDIATE

In certain situations, you might want your program to prepare and dynamically execute a statement immediately after reading it.

Suppose that you design a program to read SQL DELETE statements, similar to these, from a terminal:

```
DELETE FROM DSN8A10.EMP WHERE EMPNO = '000190'  
DELETE FROM DSN8A10.EMP WHERE EMPNO = '000220'
```

After reading a statement, the program is to run it immediately.

Recall that you must prepare (precompile and bind) static SQL statements before you can use them. You cannot prepare dynamic SQL statements in advance. The SQL statement EXECUTE IMMEDIATE causes an SQL statement to prepare and execute, dynamically, at run time.

Declaring the host variable: Before you prepare and execute an SQL statement, you can read it into a host variable. If the maximum length of the SQL statement is 32 KB, declare the host variable as a character or graphic host variable according to the following rules for the host languages:

- In assembler, PL/I, COBOL and C, you must declare a string host variable as a varying-length string.
- In Fortran, it must be a fixed-length string variable.

If the length is greater than 32 KB, you must declare the host variable as a CLOB or DBCLOB, and the maximum is 2 MB.

Example: Using a varying-length character host variable: This excerpt is from a C program that reads a DELETE statement into the host variable *dstring* and executes the statement:

```
EXEC SQL BEGIN DECLARE SECTION;
...
struct VARCHAR {
    short len;
    char s[40];
} dstring;
EXEC SQL END DECLARE SECTION;
...
/* Read a DELETE statement into the host variable dstring. */
gets(dstring);
EXEC SQL EXECUTE IMMEDIATE :dstring;
...
```

EXECUTE IMMEDIATE causes the DELETE statement to be prepared and executed immediately.

Declaring a CLOB or DBCLOB host variable: You declare CLOB and DBCLOB host variables according to certain rules.

The precompiler generates a structure that contains two elements, a 4-byte length field and a data field of the specified length. The names of these fields vary depending on the host language:

- In PL/I, assembler, and Fortran, the names are *variable_LENGTH* and *variable_DATA*.
- In COBOL, the names are *variable-LENGTH* and *variable-DATA*.
- In C, the names are *variable.LENGTH* and *variable.DATA*.

Example: Using a CLOB host variable: This excerpt is from a C program that copies an UPDATE statement into the host variable *string1* and executes the statement:

```
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS CLOB(4k) string1;
EXEC SQL END DECLARE SECTION;
...
/* Copy a statement into the host variable string1. */
strcpy(string1.data, "UPDATE DSN8610.EMP SET SALARY = SALARY * 1.1");
string1.length = 44;
EXEC SQL EXECUTE IMMEDIATE :string1;
...
```

EXECUTE IMMEDIATE causes the UPDATE statement to be prepared and executed immediately.

Related concepts:

“LOB host variable, LOB locator, and LOB file reference variable declarations” on page 757

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Dynamically executing an SQL statement by using PREPARE and EXECUTE

As an alternative to executing an SQL statement immediately after it is read, you can prepare and execute the SQL statement in two steps. This two-step method is useful when you need to execute an SQL statement multiple times with different values.

Suppose that you want to execute DELETE statements repeatedly using a list of employee numbers. Consider how you would do it if you could write the DELETE statement as a static SQL statement:

```
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
  EXEC SQL
    DELETE FROM DSN8A10.EMP WHERE EMPNO = :EMP ;
  < Read a value for EMP from the list. >
END;
```

The loop repeats until it reads an EMP value of 0.

If you know in advance that you will use only the DELETE statement and only the table DSN8A10.EMP, you can use the more efficient static SQL. Suppose further that several different tables have rows that are identified by employee numbers, and that users enter a table name as well as a list of employee numbers to delete. Although variables can represent the employee numbers, they cannot represent the table name, so you must construct and execute the entire statement dynamically. Your program must now do these things differently:

- Use parameter markers instead of host variables
- Use the PREPARE statement
- Use EXECUTE instead of EXECUTE IMMEDIATE

Parameter markers with PREPARE and EXECUTE: Dynamic SQL statements cannot use host variables. Therefore, you cannot dynamically execute an SQL statement that contains host variables. Instead, substitute a *parameter marker*, indicated by a question mark (?), for each host variable in the statement.

You can indicate to DB2 that a parameter marker represents a host variable of a certain data type by specifying the parameter marker as the argument of a CAST specification. When the statement executes, DB2 converts the host variable to the data type in the CAST specification. A parameter marker that you include in a CAST specification is called a *typed* parameter marker. A parameter marker without a CAST specification is called an *untyped* parameter marker.

Recommendation: Because DB2 can evaluate an SQL statement with typed parameter markers more efficiently than a statement with untyped parameter markers, use typed parameter markers whenever possible. Under certain circumstances you must use untyped parameter markers.

Example using parameter markers: Suppose that you want to prepare this statement:

```
DELETE FROM DSN8A10.EMP WHERE EMPNO = :EMP;
```


You need to prepare a string like this:

```
DELETE FROM DSN8A10.EMP WHERE EMPNO = CAST(? AS CHAR(6))
```

You associate host variable :EMP with the parameter marker when you execute the prepared statement. Suppose that S1 is the prepared statement. Then the EXECUTE statement looks like this:

```
EXECUTE S1 USING :EMP;
```

Using the PREPARE statement: Before you prepare an SQL statement, you can assign it to a host variable. If the length of the statement is greater than 32 KB, you must declare the host variable as a CLOB or DBCLOB.

You can think of PREPARE and EXECUTE as an EXECUTE IMMEDIATE done in two steps. The first step, PREPARE, turns a character string into an SQL statement, and then assigns it a name of your choosing.

Example using the PREPARE statement: Assume that the character host variable :DSTRING has the value "DELETE FROM DSN8A10.EMP WHERE EMPNO = ?". To prepare an SQL statement from that string and assign it the name S1, write:

```
EXEC SQL PREPARE S1 FROM :DSTRING;
```

The prepared statement still contains a parameter marker, for which you must supply a value when the statement executes. After the statement is prepared, the table name is fixed, but the parameter marker enables you to execute the same statement many times with different values of the employee number.

Using the EXECUTE statement: The EXECUTE statement executes a prepared SQL statement by naming a list of one or more host variables, one or more host variable arrays, or a host structure. This list supplies values for all of the parameter markers.

After you prepare a statement, you can execute it many times within the same unit of work. In most cases, COMMIT or ROLLBACK destroys statements prepared in a unit of work. Then, you must prepare them again before you can execute them again. However, if you declare a cursor for a dynamic statement and use the option WITH HOLD, a commit operation does not destroy the prepared statement if the cursor is still open. You can execute the statement in the next unit of work without preparing it again.

Example using the EXECUTE statement: To execute the prepared statement S1 just once, using a parameter value contained in the host variable :EMP, write:

```
EXEC SQL EXECUTE S1 USING :EMP;
```

Preparing and executing the example DELETE statement: The example in this topic began with a DO loop that executed a static SQL statement repeatedly:

```
< Read a value for EMP from the list. >  
DO UNTIL (EMP = 0);  
  EXEC SQL  
    DELETE FROM DSN8A10.EMP WHERE EMPNO = :EMP ;  
  < Read a value for EMP from the list. >  
END;
```

You can now write an equivalent example for a dynamic SQL statement:

```
< Read a statement containing parameter markers into DSTRING.>  
EXEC SQL PREPARE S1 FROM :DSTRING;  
< Read a value for EMP from the list. >
```



```
DO UNTIL (EMPNO = 0);
  EXEC SQL EXECUTE S1 USING :EMP;
  < Read a value for EMP from the list. >
END;
```

The PREPARE statement prepares the SQL statement and calls it S1. The EXECUTE statement executes S1 repeatedly, using different values for EMP.

Using more than one parameter marker: The prepared statement (S1 in the example) can contain more than one parameter marker. If it does, the USING clause of EXECUTE specifies a list of variables or a host structure. The variables must contain values that match the number and data types of parameters in S1 in the proper order. You must know the number and types of parameters in advance and declare the variables in your program, or you can use an SQLDA (SQL descriptor area).

Related concepts:

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Related tasks:

“Dynamically executing an SQL statement by using EXECUTE IMMEDIATE” on page 219

Related reference:

 [PREPARE \(DB2 SQL\)](#)

Dynamically executing a data change statement

Dynamically executing data change statements with host variable arrays is useful if you want to enter rows of data into different tables. It is also useful if you want to enter a different number of rows. The process is similar for both INSERT and MERGE statements.

For example, suppose that you want to repeatedly execute a multiple-row INSERT statement with a list of activity IDs, activity keywords, and activity descriptions that are provided by the user. You can use the following static SQL INSERT statement to insert multiple rows of data into the activity table:

```
EXEC SQL
  INSERT INTO DSN8A10.ACT
  VALUES (:hva_actno, :hva_actkwd, :hva_actdesc)
  FOR :num_rows ROWS;
```

However, if you want to enter the rows of data into different tables or enter different numbers of rows, you can construct the INSERT statement dynamically.

This topic describes the following methods that you can use to execute a data change statement dynamically:

- By using host variable arrays that contain the data to be inserted
- By using a descriptor to describe the host variable arrays that contain the data

Dynamically executing a data change statement by using host variable arrays:

To dynamically execute a data change statement by using host variable arrays, perform the following actions in your program:

1. Assign the appropriate INSERT or MERGE statement to a host variable. If needed, use the CAST specification to explicitly assign types to parameter markers that represent host variable arrays.

Example: For the activity table, the following string contains an INSERT statement that is to be prepared:

```
INSERT INTO DSN8A10.ACT
VALUES (CAST(? AS SMALLINT), CAST(? AS CHAR(6)), CAST(? AS VARCHAR(20)))
```

2. Assign any attributes for the SQL statement to a host variable.
3. Include a PREPARE statement for the SQL statement.
4. Include an EXECUTE statement with the FOR *n* ROWS clause.

Each host variable in the USING clause of the EXECUTE statement represents an array of values for the corresponding column of the target of the SQL statement. You can vary the number of rows without needing to prepare the SQL statement again.

Example: The following code prepares and executes an INSERT statement:

```
/* Copy the INSERT string into the host variable sqlstmt */
strcpy(sqlstmt, "INSERT INTO DSN8A10.ACT VALUES (CAST(? AS SMALLINT),");
strcat(sqlstmt, " CAST(? AS CHAR(6)), CAST(? AS VARCHAR(20)))");
```

```
/* Copy the INSERT attributes into the host variable attrvar */
strcpy(attrvar, "FOR MULTIPLE ROWS");
```

```
/* Prepare and execute my_insert using the host variable arrays */
EXEC SQL PREPARE my_insert ATTRIBUTES :attrvar FROM :sqlstmt;
EXEC SQL EXECUTE my_insert USING :hva1, :hva2, :hva3 FOR :num_rows ROWS;
```

Dynamically executing a data change statement by using descriptors:

You can use an SQLDA structure to specify data types and other information about the host variable arrays that contain the values to insert.

To dynamically execute a data change statement by using descriptors, perform the following actions in your program:

1. Set the following fields in the SQLDA structure for your INSERT statement.
 - SQLN
 - SQLABC
 - SQLD
 - SQLVAR
 - SQLNAME

Example: Assume that your program includes the standard SQLDA structure declaration and declarations for the program variables that point to the SQLDA structure. For C application programs, the following example code sets the SQLDA fields:

```
strcpy(sqldaptr->sqldaid, "SQLDA");
sqldaptr->sqldabc = 192; /* number of bytes of storage allocated
for the SQLDA */
sqldaptr->sqln = 4; /* number of SQLVAR
occurrences */
sqldaptr->sqld = 4;
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0])); /* Point
to first SQLVAR */
```

```

varptr->sqltype = 500;                               /* data
type SMALLINT */
varptr->sqllen = 2;
varptr->sqldata = (char *) hva1;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\x00\x00\x00\x00\x00\x01\x00\x14", varptr->sqlname.length);
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]) + 1); /* Point
to next SQLVAR */
varptr->sqltype = 452;                               /* data
type CHAR(6) */
varptr->sqllen = 6;
varptr->sqldata = (char *) hva2;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\x00\x00\x00\x00\x00\x01\x00\x14", varptr->sqlname.length);
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]) + 2); /* Point
to next SQLVAR */
varptr->sqltype = 448;                               /* data type
VARCHAR(20) */
varptr->sqllen = 20;
varptr->sqldata = (char *) hva3;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\x00\x00\x00\x00\x00\x01\x00\x14", varptr->sqlname.length);

```

The SQLDA structure has the following fields:

- SQLDABC indicates the number of bytes of storage that are allocated for the SQLDA. The storage includes a 16-byte header and 44 bytes for each SQLVAR field. The value is $SQLN \times 44 + 16$, or 192 for this example.
 - SQLN is the number of SQLVAR occurrences, plus one for use by DB2 for the host variable that contains the number n in the FOR n ROWS clause.
 - SQLD is the number of variables in the SQLDA that are used by DB2 when processing the INSERT statement.
 - An SQLVAR occurrence specifies the attributes of an element of a host variable array that corresponds to a value provided for a target column of the INSERT. Within each SQLVAR:
 - SQLTYPE indicates the data type of the elements of the host variable array.
 - SQLLEN indicates the length of a single element of the host variable array.
 - SQLDATA points to the corresponding host variable array. Assume that your program allocates the dynamic variable arrays hva1, hva2, and hva3.
 - SQLNAME has two parts: the LENGTH and the DATA. The LENGTH is 8. The first two bytes of the DATA field is X'0000'. Bytes 5 and 6 of the DATA field are a flag indicating whether the variable is an array or a FOR n ROWS value. Bytes 7 and 8 are a two-byte binary integer representation of the dimension of the array.
2. Assign the appropriate INSERT or MERGE statement to a host variable.

Example: The following string contains an INSERT statement that is to be prepared:

```
INSERT INTO DSN8A10.ACT VALUES (?, ?, ?)
```

3. Assign any attributes for the SQL statement to a host variable.
4. Include a PREPARE statement for the SQL statement.
5. Include an EXECUTE statement with the FOR n ROWS clause. The host variable in the USING clause of the EXECUTE statement names the SQLDA that describes the parameter markers in the INSERT statement.

Example: The following code prepares and executes an INSERT statement:

```

/* Copy the INSERT string into the host variable sqlstmt */
strcpy(sqlstmt, "INSERT INTO DSN8A10.ACT VALUES (?, ?, ?)");

/* Copy the INSERT attributes into the host variable attrvar */
strcpy(attrvar, "FOR MULTIPLE ROWS");

/* Prepare and execute my_insert using the descriptor */
EXEC SQL PREPARE my_insert ATTRIBUTES :attrvar FROM :sqlstmt;
EXEC SQL EXECUTE my_insert USING DESCRIPTOR :*sqldaptr FOR :num_rows ROWS;

```

Related concepts:

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

Related tasks:

“Including dynamic SQL for varying-list SELECT statements in your program” on page 202

Related reference:

 [SQLTYPE and SQLLEN \(DB2 SQL\)](#)

Dynamically executing a statement with parameter markers by using the SQLDA

Your program can get data type information about parameter markers by asking DB2 to set the fields in the SQLDA.

Before you dynamically execute a statement with parameter markers, allocate an SQLDA with enough instances of SQLVAR to represent all parameter markers in the SQL statement.

To dynamically execute a statement with parameter markers by using the SQLDA:

1. Include in your program a DESCRIBE INPUT statement that specifies the prepared SQL statement and the name of an appropriate SQLDA.
DB2 puts the requested parameter marker information in the SQLDA.
2. Code the application in the same way as any other application in which you execute a prepared statement by using an SQLDA. First, obtain the addresses of the input host variables and their indicator variables and insert those addresses into the SQLDATA and SQLIND fields. Then, execute the prepared SQL statement.

Suppose that you want to execute the following statement dynamically:

```
DELETE FROM DSN8A10.EMP WHERE EMPNO = ?
```

You can use the following code to set up an SQLDA, obtain parameter information by using the DESCRIBE INPUT statement, and execute the statement:

```

SQLDAPTR=ADDR(INSQLDA);           /* Get pointer to SQLDA      */
SQLDAID='SQLDA';                  /* Fill in SQLDA eye-catcher */
SQLDABC=LENGTH(INSQLDA);          /* Fill in SQLDA length     */
SQLN=1;                            /* Fill in number of SQLVARs */
SQLD=0;                             /* Initialize # of SQLVARs used */
DO IX=1 TO SQLN;                   /* Initialize the SQLVAR     */
    SQLTYPE(IX)=0;
    SQLLEN(IX)=0;
    SQLNAME(IX)='';
END;

```

```

SQLSTMT='DELETE FROM DSN8A10.EMP WHERE EMPNO = ?';
EXEC SQL PREPARE SQLOBJ FROM SQLSTMT;
EXEC SQL DESCRIBE INPUT SQLOBJ INTO :INSQLDA;
SQLDATA(1)=ADDR(HVEMP);           /* Get input data address      */
SQLIND(1)=ADDR(HVEMPIND);         /* Get indicator address      */
EXEC SQL EXECUTE SQLOBJ USING DESCRIPTOR :INSQLDA;

```

Related concepts:

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Related tasks:

“Defining SQL descriptor areas” on page 173

Related reference:

 DESCRIBE INPUT (DB2 SQL)

Checking the execution of SQL statements

After executing an SQL statement, your program should check for any errors codes before you commit the data and handle the errors that they represent.

You can check the execution of SQL statements in one of the following ways:

- By displaying specific fields in the SQLCA.
- By testing SQLCODE or SQLSTATE for specific values.
- By using the WHENEVER statement in your application program.
- By testing indicator variables to detect numeric errors.
- By using the GET DIAGNOSTICS statement in your application program to return all the condition information that results from the execution of an SQL statement.
- By calling DSNTIAR to display the contents of the SQLCA.

Related concepts:

“Arithmetic and conversion errors” on page 240

Related tasks:

“Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler” on page 253

“Defining the SQL communications area, SQLSTATE, and SQLCODE in C” on page 273

“Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL” on page 323

“Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran” on page 395

“Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I” on page 407

“Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX” on page 437

“Displaying SQLCA fields by calling DSNTIAR” on page 229

Checking the execution of SQL statements by using the SQLCA

One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with DB2.

If you use the SQLCA, include the necessary instructions to display information that is contained in the SQLCA in your application program. Alternatively, you can use the GET DIAGNOSTICS statement, which is an SQL standard, to diagnose problems.

- When DB2 processes an SQL statement, it places return codes that indicate the success or failure of the statement execution in SQLCODE and SQLSTATE.
- When DB2 processes a FETCH statement, and the FETCH is successful, the contents of SQLERRD(3) in the SQLCA is set to the number of returned rows.
- When DB2 processes a multiple-row FETCH statement, the contents of SQLCODE is set to +100 if the last row in the table has been returned with the set of rows.
- When DB2 processes an UPDATE, INSERT, or DELETE statement, and the statement execution is successful, the contents of SQLERRD(3) in the SQLCA is set to the number of rows that are updated, inserted, or deleted.
- When DB2 processes a TRUNCATE statement and the statement execution is successful, SQLERRD(3) in the SQLCA is set to -1. The number of rows that are deleted is not returned.
- If SQLWARN0 contains **W**, DB2 has set at least one of the SQL warning flags (SQLWARN1 through SQLWARNA):
 - SQLWARN1 contains **N** for non-scrollable cursors and **S** for scrollable cursors after an OPEN CURSOR or ALLOCATE CURSOR statement.
 - SQLWARN4 contains **I** for insensitive scrollable cursors, **S** for sensitive static scrollable cursors, and **D** for sensitive dynamic scrollable cursors, after an OPEN CURSOR or ALLOCATE CURSOR statement, or blank if the cursor is not scrollable.
 - SQLWARN5 contains a character value of **1** (read only), **2** (read and delete), or **4** (read, delete, and update) to indicate the operation that is allowed on the result table of the cursor.

Related tasks:

“Accessing data by using a rowset-positioned cursor” on page 731

“Checking the execution of SQL statements by using SQLCODE and SQLSTATE” on page 232

“Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler” on page 253

“Defining the SQL communications area, SQLSTATE, and SQLCODE in C” on page 273

“Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL” on page 323

“Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran” on page 395

“Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I” on page 407

“Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX” on page 437

Related reference:

 [Description of SQLCA fields \(DB2 SQL\)](#)

Displaying SQLCA fields by calling DSNTIAR

If you use the SQLCA to check whether an SQL statement executed successfully, your program needs to read the data in the appropriate SQLCA fields. One easy way to read these fields is to use the assembler subroutine DSNTIAR.

You should check for errors codes before you commit data, and handle the errors that they represent. The assembler subroutine DSNTIAR helps you to obtain a formatted form of the SQLCA and a text message based on the SQLCODE field of the SQLCA. You can retrieve this same message text by using the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR.

DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. Each time you use DSNTIAR, it overwrites any previous messages in the message output area. You should move or print the messages before using DSNTIAR again, and before the contents of the SQLCA change, to get an accurate view of the SQLCA.

DSNTIAR expects the SQLCA to be in a certain format. If your application modifies the SQLCA format before you call DSNTIAR, the results are unpredictable.

DSNTIAR:

The assembler subroutine DSNTIAR helps you to obtain a formatted form of the SQLCA and a text message that is based on the SQLCODE field of the SQLCA.

DSNTIAR can run either above or below the 16-MB line of virtual storage. The DSNTIAR object module that comes with DB2 has the attributes AMODE(31) and RMODE(ANY). At installation time, DSNTIAR links as AMODE(31) and RMODE(ANY). DSNTIAR runs in 31-bit mode if any of the following conditions is true:

- DSNTIAR is linked with other modules that also have the attributes AMODE(31) and RMODE(ANY).
- DSNTIAR is linked into an application that specifies the attributes AMODE(31) and RMODE(ANY) in its link-edit JCL.
- An application loads DSNTIAR.

When loading DSNTIAR from another program, be careful how you branch to DSNTIAR. For example, if the calling program is in 24-bit addressing mode and DSNTIAR is loaded above the 16-MB line, you cannot use the assembler BALR instruction or CALL macro to call DSNTIAR, because they assume that DSNTIAR is in 24-bit mode. Instead, you must use an instruction that is capable of branching into 31-bit mode, such as BASSM.

You can dynamically link (load) and call DSNTIAR directly from a language that does not handle 31-bit addressing. To do this, link a second version of DSNTIAR with the attributes AMODE(24) and RMODE(24) into another load module library.

Alternatively, you can write an intermediate assembler language program that calls DSNTIAR in 31-bit mode and then call that intermediate program in 24-bit mode from your application.

For more information on the allowed and default AMODE and RMODE settings for a particular language, see the application programming guide for that language. For details on how the attributes AMODE and RMODE of an application are determined, see the linkage editor and loader user's guide for the language in which you have written the application.

Defining a message output area:

If a program calls DSNTIAR, the program must allocate enough storage in the message output area to hold all of the message text that DSNTIAR returns.

You will probably need no more than 10 lines, 80-bytes each, for your message output area. An application program can have only one message output area.

You must define the message output area in VARCHAR format. In this varying character format, a 2-byte length field precedes the data. The length field indicates to DSNTIAR how many total bytes are in the output message area; the minimum length of the output area is 240-bytes.

The following figure shows the format of the message output area, where *length* is the 2-byte total length field, and the length of each line matches the logical record length (*lrecl*) you specify to DSNTIAR.

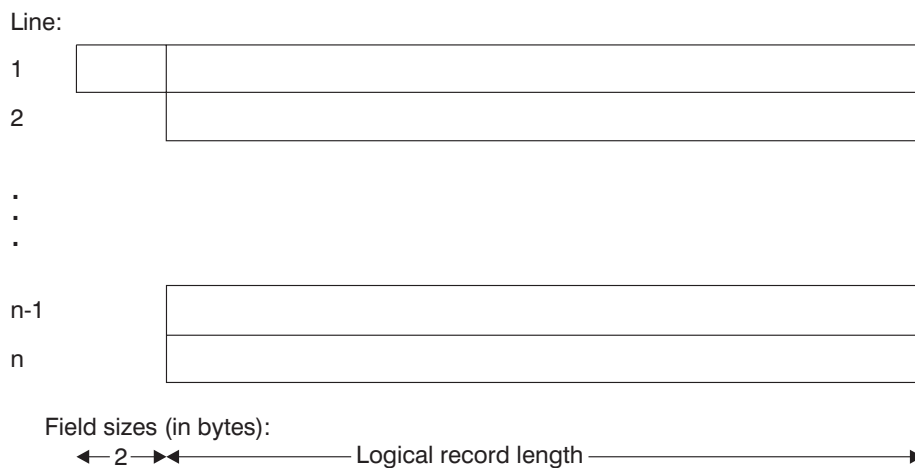


Figure 21. Format of the message output area

When you call DSNTIAR, you must name an SQLCA and an output message area in the DSNTIAR parameters. You must also provide the logical record length (*lrecl*) as a value between 72 and 240 bytes. DSNTIAR assumes the message area contains fixed-length records of length *lrecl*.

DSNTIAR places up to 10 lines in the message area. If the text of a message is longer than the record length you specify on DSNTIAR, the output message splits into several records, on word boundaries if possible. The split records are indented. All records begin with a blank character for carriage control. If you have more lines than the message output area can contain, DSNTIAR issues a return code of 4. A completely blank record marks the end of the message output area.

Possible return codes from DSNTIAR:

The assembler subroutine DSNTIAR helps your program read the information in the SQLCA. The subroutine also returns its own return code.

Code Meaning

0	Successful execution.
4	More data available than could fit into the provided message area.
8	Logical record length not between 72 and 240, inclusive.
12	Message area not large enough. The message length was 240 or greater.
16	Error in TSO message routine.
20	Module DSNTIA1 could not be loaded.
24	SQLCA data error.

A scenario for using DSNTIAR:

You can use the assembler subroutine DSNTIAR to generate the error message text in the SQLCA.

Suppose you want your DB2 COBOL application to check for deadlocks and timeouts, and you want to make sure your cursors are closed before continuing. You use the statement `WHENEVER SQLERROR` to transfer control to an error routine when your application receives a negative `SQLCODE`.

In your error routine, you write a section that checks for `SQLCODE -911` or `-913`. You can receive either of these `SQLCODE`s when a deadlock or timeout occurs. When one of these errors occurs, the error routine closes your cursors by issuing the statement:

```
EXEC SQL CLOSE cursor-name
```

An `SQLCODE` of 0 or -501 resulting from that statement indicates that the close was successful.

To use DSNTIAR to generate the error message text, first follow these steps:

1. Choose a logical record length (*lrecl*) of the output lines. For this example, assume *lrecl* is 72 (to fit on a terminal screen) and is stored in the variable named `ERROR-TEXT-LEN`.
2. Define a message area in your COBOL application. Assuming you want an area for up to 10 lines of length 72, you should define an area of 720 bytes, plus a 2-byte area that specifies the total length of the message output area.

```
01 ERROR-MESSAGE.  
    02 ERROR-LEN   PIC S9(4)  COMP VALUE +720.  
    02 ERROR-TEXT PIC X(72)  OCCURS 10 TIMES  
                                INDEXED BY ERROR-INDEX.  
77 ERROR-TEXT-LEN PIC S9(9)  COMP VALUE +72.
```

For this example, the name of the message area is `ERROR-MESSAGE`.

3. Make sure you have an SQLCA. For this example, assume the name of the SQLCA is `SQLCA`.

To display the contents of the SQLCA when `SQLCODE` is 0 or -501, call DSNTIAR after the SQL statement that produces `SQLCODE` 0 or -501:

```
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

You can then print the message output area just as you would any other variable. Your message might look like this:

```
DSNT408I SQLCODE = -501, ERROR: THE CURSOR IDENTIFIED IN A FETCH OR  
CLOSE STATEMENT IS NOT OPEN  
DSNT418I SQLSTATE = 24501 SQLSTATE RETURN CODE  
DSNT415I SQLERRP = DSNXERT SQL PROCEDURE DETECTING ERROR  
DSNT416I SQLERRD = -315 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION  
DSNT416I SQLERRD = X'FFFFFFEC5' X'00000000' X'00000000'  
X'FFFFFFF' X'00000000' X'00000000' SQL DIAGNOSTIC  
INFORMATION
```

Checking the execution of SQL statements by using SQLCODE and SQLSTATE

Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code. Portable applications should use SQLSTATE instead of SQLCODE, although SQLCODE values can provide additional DB2-specific information about an SQL error or warning.

SQLCODE:

DB2 returns the following codes in SQLCODE:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

SQLCODE 100 indicates that no data was found.

The meaning of SQLCODEs other than 0 and 100 varies with the particular product implementing SQL.

SQLSTATE: SQLSTATE enables an application program to check for errors in the same way for different IBM database management systems.

Using SQLCODE and SQLSTATE:

An advantage to using the SQLCODE field is that it can provide more specific information than the SQLSTATE. Many of the SQLCODEs have associated tokens in the SQLCA that indicate, for example, which object incurred an SQL error. However, an SQL standard application uses only SQLSTATE.

You can declare SQLCODE and SQLSTATE (SQLCOD and SQLSTA in Fortran) as stand-alone host variables. If you specify the STDSQL(YES) precompiler option, these host variables receive the return codes, and you should not include an SQLCA in your program.

Related tasks:

“Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler” on page 253

“Defining the SQL communications area, SQLSTATE, and SQLCODE in C” on page 273

“Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL” on page 323

“Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran” on page 395

“Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I” on page 407

“Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX” on page 437

Related reference:

 [SQLSTATE values and common error codes \(DB2 Codes\)](#)

Checking the execution of SQL statements by using the WHENEVER statement

The WHENEVER statement causes DB2 to check the SQLCA and continue processing your program. If an error, exception, or warning occurs, DB2 branches to another area in your program. The condition handling area of your program can then examine the SQLCODE or SQLSTATE to react specifically to the error or exception.

The WHENEVER statement is not supported for REXX.

The WHENEVER statement enables you to specify what to do if a general condition is true. You can specify more than one WHENEVER statement in your program. When you do this, the first WHENEVER statement applies to all subsequent SQL statements in the source program until the next WHENEVER statement.

The WHENEVER statement looks like this:

```
EXEC SQL
    WHENEVER condition action
END-EXEC
```

The *condition* of the WHENEVER statement is one of these three values:

SQLWARNING

Indicates what to do when SQLWARN0 = W or SQLCODE contains a positive value other than 100. DB2 can set SQLWARN0 for several reasons—for example, if a column value is truncated when moved into a host variable. Your program might not regard this as an error.

SQLERROR

Indicates what to do when DB2 returns an error code as the result of an SQL statement (SQLCODE < 0).

NOT FOUND

Indicates what to do when DB2 cannot find a row to satisfy your SQL statement or when there are no more rows to fetch (SQLCODE = 100).

The *action* of the WHENEVER statement is one of these two values:

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, preceded by an optional colon. The form of the token depends on the host language. In COBOL, for example, it can be *section-name* or an unqualified *paragraph-name*.

The WHENEVER statement must precede the first SQL statement it is to affect. However, if your program checks SQLCODE directly, you must check SQLCODE after each SQL statement.

Related concepts:

Chapter 9, “Coding SQL statements in REXX application programs,” on page 437

Related reference:

 WHENEVER (DB2 SQL)

Checking the execution of SQL statements by using the GET DIAGNOSTICS statement

One way to check whether an SQL statement executed successfully is to ask DB2 to return the diagnostic information about the last executed SQL statement.

You can use the GET DIAGNOSTICS statement to return diagnostic information about the last SQL statement that was executed. You can request individual items of diagnostic information from the following groups of items:

- Statement items, which contain information about the SQL statement as a whole
- Condition items, which contain information about each error or warning that occurred during the execution of the SQL statement
- Connection items, which contain information about the SQL statement if it was a CONNECT statement

In addition to requesting individual items, you can request that GET DIAGNOSTICS return ALL diagnostic items that are set during the execution of the last SQL statement as a single string.

In SQL procedures, you can also retrieve diagnostic information by using handlers. Handlers tell the procedure what to do if a particular error occurs.

Use the GET DIAGNOSTICS statement to handle multiple SQL errors that might result from the execution of a single SQL statement. First, check SQLSTATE (or SQLCODE) to determine whether diagnostic information should be retrieved by using GET DIAGNOSTICS. This method is especially useful for diagnosing problems that result from a multiple-row INSERT that is specified as NOT ATOMIC CONTINUE ON SQLEXCEPTION and multiple row MERGE statements.

Even if you use only the GET DIAGNOSTICS statement in your application program to check for conditions, you must either include the instructions required to use the SQLCA or you must declare SQLSTATE (or SQLCODE) separately in your program.

When you use the GET DIAGNOSTICS statement, you assign the requested diagnostic information to host variables. Declare each target host variable with a data type that is compatible with the data type of the requested item.

To retrieve condition information, you must first retrieve the number of condition items (that is, the number of errors and warnings that DB2 detected during the execution of the last SQL statement). The number of condition items is at least one. If the last SQL statement returned SQLSTATE '00000' (or SQLCODE 0), the number of condition items is one.

Example: Using GET DIAGNOSTICS with multiple-row INSERT:

You want to display diagnostic information for each condition that might occur during the execution of a multiple-row INSERT statement in your application program. You specify the INSERT statement as NOT ATOMIC CONTINUE ON SQLEXCEPTION, which means that execution continues regardless of the failure of any single-row insertion. DB2 does not insert the row that was processed at the time of the error.

In the following example, the first GET DIAGNOSTICS statement returns the number of rows inserted and the number of conditions returned. The second GET DIAGNOSTICS statement returns the following items for each condition: SQLCODE, SQLSTATE, and the number of the row (in the rowset that was being inserted) for which the condition occurred.

```
EXEC SQL BEGIN DECLARE SECTION;
    long row_count, num_condns, i;
    long ret_sqlcode, row_num;
    char ret_sqlstate[6];
    ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
    INSERT INTO DSN8A10.ACT
        (ACTNO, ACTKWD, ACTDESC)
        VALUES (:hva1, :hva2, :hva3)
        FOR 10 ROWS
        NOT ATOMIC CONTINUE ON SQLEXCEPTION;

EXEC SQL GET DIAGNOSTICS
    :row_count = ROW_COUNT, :num_condns = NUMBER;
printf("Number of rows inserted = %d\n", row_count);

for (i=1; i<=num_condns; i++) {
    EXEC SQL GET DIAGNOSTICS CONDITION :i
        :ret_sqlcode = DB2_RETURNED_SQLCODE,
        :ret_sqlstate = RETURNED_SQLSTATE,
        :row_num = DB2_ROW_NUMBER;
    printf("SQLCODE = %d, SQLSTATE = %s, ROW NUMBER = %d\n",
        ret_sqlcode, ret_sqlstate, row_num);
}
```

In the activity table, the ACTNO column is defined as SMALLINT. Suppose that you declare the host variable array hva1 as an array with data type long, and you populate the array so that the value for the fourth element is 32768.

If you check the SQLCA values after the INSERT statement, the value of SQLCODE is equal to 0, the value of SQLSTATE is '00000', and the value of SQLERRD(3) is 9 for the number of rows that were inserted. However, the INSERT statement specified that 10 rows were to be inserted.

The GET DIAGNOSTICS statement provides you with the information that you need to correct the data for the row that was not inserted. The printed output from your program looks like this:

```
Number of rows inserted = 9
SQLCODE = -302, SQLSTATE = 22003, ROW NUMBER = 4
```

The value 32768 for the input variable is too large for the target column ACTNO. You can print the MESSAGE_TEXT condition item.

Retrieving statement and condition items:

When you use the GET DIAGNOSTICS statement, you assign the requested diagnostic information to host variables. Declare each target host variable with a data type that is compatible with the data type of the requested item.

To retrieve condition information, you must first retrieve the number of condition items (that is, the number of errors and warnings that DB2 detected during the execution of the last SQL statement). The number of condition items is at least one. If the last SQL statement returned SQLSTATE '00000' (or SQLCODE 0), the number of condition items is one.

Related concepts:

“Handlers in an SQL procedure” on page 578

Related reference:

“Data types for GET DIAGNOSTICS items”

 GET DIAGNOSTICS (DB2 SQL)

Related information:

 -302 (DB2 Codes)

Data types for GET DIAGNOSTICS items

You can use the GET DIAGNOSTICS statement to request information about the statement, condition, and connection for the last SQL statement that was executed. You must declare each target host variable with a data type that is compatible with the data type of the requested item.

The following tables specify the data types for the statement, condition, and connection information items that you can request by using the GET DIAGNOSTICS statement.

Table 47. Data types for GET DIAGNOSTICS items that return statement information

Item	Description	Data type
DB2_GET_DIAGNOSTICS_DIAGNOSTICS	After a GET DIAGNOSTICS statement, if any error or warning occurred, this item contains all of the diagnostics as a single string.	VARCHAR(32672)
DB2_LAST_ROW	After a multiple-row FETCH statement, this item contains a value of +100 if the last row in the table is in the rowset that was returned.	INTEGER
DB2_NUMBER_PARAMETER_MARKERS	After a PREPARE statement, this item contains the number of parameter markers in the prepared statement.	INTEGER
DB2_NUMBER_RESULT_SETS	After a CALL statement that invokes a stored procedure, this item contains the number of result sets that are returned by the procedure.	INTEGER

Table 47. Data types for GET DIAGNOSTICS items that return statement information (continued)

Item	Description	Data type
DB2_NUMBER_ROWS	After an OPEN or FETCH statement for which the size of the result table is known, this item contains the number of rows in the result table. After a PREPARE statement, this item contains the estimated number of rows in the result table for the prepared statement. For SENSITIVE DYNAMIC cursors, this item contains the approximate number of rows. Otherwise, or if the server only returns an SQLCA, the value zero is returned.	DECIMAL(31,0)
DB2_RETURN_STATUS	After a CALL statement that invokes an SQL procedure, this item contains the return status if the procedure contains a RETURN statement.	INTEGER
DB2_SQL_ATTR_CURSOR_HOLD	After an ALLOCATE or OPEN statement, this item indicates whether the cursor can be held open across multiple units of work (Y or N).	CHAR(1)
DB2_SQL_ATTR_CURSOR_ROWSET	After an ALLOCATE or OPEN statement, this item indicates whether the cursor can use rowset positioning (Y or N).	CHAR(1)
DB2_SQL_ATTR_CURSOR_SCROLLABLE	After an ALLOCATE or OPEN statement, this item indicates whether the cursor is scrollable (Y or N).	CHAR(1)
DB2_SQL_ATTR_CURSOR_SENSITIVITY	After an ALLOCATE or OPEN statement, this item indicates whether the cursor shows updates made by other processes (sensitivity I or S).	CHAR(1)
DB2_SQL_ATTR_CURSOR_TYPE	After an ALLOCATE or OPEN statement, this item indicates whether the cursor is forward (F), declared static (S for INSENSITIVE or SENSITIVE STATIC, or dynamic (D for SENSITIVE DYNAMIC).	CHAR(1)
MORE	After any SQL statement, this item indicates whether some condition items were discarded because of insufficient storage (Y or N).	CHAR(1)
NUMBER	After any SQL statement, this item contains the number of condition items. If no warning or error occurred, or if no previous SQL statement has been executed, the number that is returned is 1.	INTEGER

Table 47. Data types for GET DIAGNOSTICS items that return statement information (continued)

Item	Description	Data type
ROW_COUNT	After an insert, update, delete, or fetch, this item contains the number of rows that are deleted, inserted, updated, or fetched. After PREPARE, this item contains the estimated number of result rows in the prepared statement. After TRUNCATE, it contains -1.	DECIMAL(31,0)

Table 48. Data types for GET DIAGNOSTICS items that return condition information

Item	Description	Data type
CATALOG_NAME	This item contains the server name of the table that owns a constraint that caused an error, or that caused an access rule or check violation.	VARCHAR(128)
CONDITION_NUMBER	This item contains the number of the condition.	INTEGER
CURSOR_NAME	This item contains the name of a cursor in an invalid cursor state.	VARCHAR(128)
DB2_ERROR_CODE1	This item contains an internal error code.	INTEGER
DB2_ERROR_CODE2	This item contains an internal error code.	INTEGER
DB2_ERROR_CODE3	This item contains an internal error code.	INTEGER
DB2_ERROR_CODE4	This item contains an internal error code.	INTEGER
DB2_INTERNAL_ERROR_POINTER	For some errors, this item contains a negative value that is an internal error pointer.	INTEGER
DB2_MESSAGE_ID	This item contains the message ID that corresponds to the message that is contained in the MESSAGE_TEXT diagnostic item.	CHAR(10)
DB2_MODULE_DETECTING_ERROR	After any SQL statement, this item indicates which module detected the error.	CHAR(8)
DB2_ORDINAL_TOKEN_n	After any SQL statement, this item contains the <i>n</i> th token, where <i>n</i> is a value from 1 to 100.	VARCHAR(515)
DB2_REASON_CODE	After any SQL statement, this item contains the reason code for errors that have a reason code token in the message text.	INTEGER
DB2_RETURNED_SQLCODE	After any SQL statement, this item contains the SQLCODE for the condition.	INTEGER
DB2_ROW_NUMBER	After any SQL statement that involves multiple rows, this item contains the row number on which DB2 detected the condition.	DECIMAL(31,0)
DB2_TOKEN_COUNT	After any SQL statement, this item contains the number of tokens available for the condition.	INTEGER
MESSAGE_TEXT	After any SQL statement, this item contains the message text associated with the SQLCODE.	VARCHAR(32672)

Table 48. Data types for GET DIAGNOSTICS items that return condition information (continued)

Item	Description	Data type
RETURNED_SQLSTATE	After any SQL statement, this item contains the SQLSTATE for the condition.	CHAR(5)
SERVER_NAME	After a CONNECT, DISCONNECT, or SET CONNECTION statement, this item contains the name of the server specified in the statement.	VARCHAR(128)

Table 49. Data types for GET DIAGNOSTICS items that return connection information

Item	Description	Data type
DB2_AUTHENTICATION_TYPE	This item contains the authentication type (S, C, D, E, or blank).	CHAR(1)
DB2_AUTHORIZATION_ID	This item contains the authorization ID that is used by the connected server.	VARCHAR(128)
DB2_CONNECTION_STATE	This item indicates whether the connection is unconnected (-1), local (0), or remote (1).	INTEGER
DB2_CONNECTION_STATUS	This item indicates whether updates can be committed for the current unit of work (1 for Yes, 2 for No).	INTEGER
DB2_ENCRYPTION_TYPE	This item contains one of the following values that indicates the level of encryption for the connection: A Only the authentication tokens (authid and password) are encrypted D All of the data for the connection is encrypted	CHAR(1)
DB2_SERVER_CLASS_NAME	After a CONNECT or SET CONNECTION statement, this item contains the DB2 server class name.	VARCHAR(128)
DB2_PRODUCT_ID	This item contains the DB2 product signature.	VARCHAR(8)

Related reference:

 GET DIAGNOSTICS (DB2 SQL)

Handling SQL error codes

You can use the subroutine DSNTIAR or the GET DIAGNOSTICS statement to convert an SQL return code into a text message.

To handle SQL error codes:

Take action based on the programming language that you use.

Related concepts:

“SQL statements in assembler programs” on page 266

“SQL statements in C programs” on page 307

“SQL statements in COBOL programs” on page 357

“SQL statements in Fortran programs” on page 403

“SQL statements in PL/I programs” on page 427

“SQL statements in REXX programs” on page 439

Arithmetic and conversion errors

You can track arithmetic and conversion errors by using indicator variables. An indicator variable contains a small integer value that indicates some information about the associated host variable.

Numeric or character conversion errors or arithmetic expression errors can set an indicator variable to -2. For example, division by zero and arithmetic overflow do not necessarily halt the execution of a SELECT statement. If you use indicator variables and an error occurs in the SELECT list, the statement can continue to execute and return good data for rows in which the error does not occur.

For rows in which a conversion or arithmetic expression error does occur, the indicator variable indicates that one or more selected items have no meaningful value. The indicator variable flags this error with a -2 for the affected host variable and an SQLCODE of +802 (SQLSTATE '01519') in the SQLCA.

Writing applications that enable users to create and modify tables

You can write a DB2 application that enables users to create new tables, add columns to them, increase the length of columns, rearrange the columns, and delete columns.

To create new tables:

- Use the CREATE TABLE statement.

To add columns or increase the length of columns:

- Use the ALTER TABLE statement with the ADD COLUMN clause or the ALTER COLUMN clause. Added columns initially contain either the null value or a default value. Both CREATE TABLE and ALTER TABLE, like any data definition statement, are relatively expensive to execute. Also consider the effects of locks.

To rearrange or delete columns:

- Drop the table and create the table again, with the columns you want, in the order you want. Consider creating a view on the table, which includes only the columns that you want, in the order that you want, as an alternative to redefining the table.

Related tasks:

“Including dynamic SQL in your program” on page 193

Related reference:

 ALTER TABLE (DB2 SQL)

 CREATE TABLE (DB2 SQL)

 CREATE VIEW (DB2 SQL)

Saving SQL statements that are translated from user requests

If your program translates requests from users into SQL statements and allows users to save their requests, your program can improve performance by saving those translated statements.

A program translates requests from users into SQL statements before executing them, and users can save a request.

To save the corresponding SQL statement:

Save the corresponding SQL statements in a table with a column having a data type of VARCHAR(*n*), where *n* is the maximum length of any SQL statement. You must save the source SQL statements, not the prepared versions. That means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your program prepares an SQL statement from a character string and executes it dynamically.

Related tasks:

“Including dynamic SQL in your program” on page 193

XML data in embedded SQL applications

Embedded SQL applications that are written in assembler language, C, C++, COBOL, or PL/I can update and retrieve data in XML columns.

In embedded SQL applications, you can:

- Store an entire XML document in an XML column using INSERT or UPDATE statements.
- Retrieve an entire XML document from an XML column using SELECT statements.
- Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function within a SELECT or FETCH statement, to retrieve the sequence into a textual XML string in the database, and then retrieve the data into an application variable.

Recommendation: Follow these guidelines when you write embedded SQL applications:

- Avoid using the XMLPARSE and XMLSERIALIZE functions.
Let DB2 do the conversions between the external and internal XML formats implicitly.
- Use XML host variables for input and output.
Doing so allows DB2 to process values as XML data instead of character or binary string data. If the application cannot use XML host variables, it should use binary string host variables to minimize character conversion issues.
- Avoid character conversion by using UTF-8 host variables for input and output of XML values whenever possible.

Host variable data types for XML data in embedded SQL applications

DB2 provides XML host variable types for assembler, C, C++, COBOL, and PL/I.

Those types are:

- XML AS BLOB
- XML AS CLOB
- XML AS DBCLOB
- XML AS BLOB_FILE (C, C++, or PL/I) or XML AS BLOB-FILE (COBOL)
- XML AS CLOB_FILE (C, C++, or PL/I) or XML AS CLOB-FILE (COBOL)
- XML AS DBCLOB_FILE (C, C++, or PL/I) or XML AS DBCLOB-FILE (COBOL)

The XML host variable types are compatible only with the XML column data type.

You can use BLOB, CLOB, DBCLOB, CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY host variables to update XML columns. You can convert the host variable data types to the XML type using the XMLPARSE function, or you can let the DB2 database server perform the conversion implicitly.

You can use BLOB, CLOB, DBCLOB, CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY host variables to retrieve data from XML columns. You can convert the XML data to the host variable type using the XMLSERIALIZE function, or you can let the DB2 database server perform the conversion implicitly.

The following examples show you how to declare XML host variables in each supported language. In each table, the left column contains the declaration that you code in your application program. The right column contains the declaration that DB2 generates.

Declarations of XML host variables in assembler

The following table shows assembler language declarations for some typical XML types.

Table 50. Example of assembler XML variable declarations

You declare this variable	DB2 generates this variable
BLOB_XML SQL TYPE IS XML AS BLOB IM	BLOB_XML DS 0FL4 BLOB_XML_LENGTH DS FL4 BLOB_XML_DATA DS CL65535 ¹ ORG *(983041)
CLOB_XML SQL TYPE IS XML AS CLOB 40000K	CLOB_XML DS 0FL4 CLOB_XML_LENGTH DS FL4 CLOB_XML_DATA DS CL65535 ¹ ORG *(40894465)
DBCLOB_XML SQL TYPE IS XML AS DBCLOB 4000K	DBCLOB_XML DS 0FL4 DBCLOB_XML_LENGTH DS FL4 DBCLOB_XML_DATA DS GL65534 ² ORG *(4030466)
BLOB_XML_FILE SQL TYPE IS XML AS BLOB_FILE	BLOB_XML_FILE DS 0FL4 BLOB_XML_FILE_NAME_LENGTH DS FL4 BLOB_XML_FILE_DATA_LENGTH DS FL4 BLOB_XML_FILE_FILE_OPTIONS DS FL4 BLOB_XML_FILE_NAME DS CL255
CLOB_XML_FILE SQL TYPE IS XML AS CLOB_FILE	CLOB_XML_FILE DS 0FL4 CLOB_XML_FILE_NAME_LENGTH DS FL4 CLOB_XML_FILE_DATA_LENGTH DS FL4 CLOB_XML_FILE_FILE_OPTIONS DS FL4 CLOB_XML_FILE_NAME DS CL255
DBCLOB_XML_FILE SQL TYPE IS XML AS DBCLOB_FILE	DBCLOB_XML_FILE DS 0FL4 DBCLOB_XML_FILE_NAME_LENGTH DS FL4 DBCLOB_XML_FILE_DATA_LENGTH DS FL4 DBCLOB_XML_FILE_FILE_OPTIONS DS FL4 DBCLOB_XML_FILE_NAME DS CL255

Table 50. Example of assembler XML variable declarations (continued)

You declare this variable	DB2 generates this variable
Notes:	
1. Because assembler language allows character declarations of no more than 65535 bytes, DB2 separates the host language declarations for XML AS BLOB and XML AS CLOB host variables that are longer than 65535 bytes into two parts.	
2. Because assembler language allows graphic declarations of no more than 65534 bytes, DB2 separates the host language declarations for XML AS DBCLOB host variables that are longer than 65534 bytes into two parts.	

Declarations of XML host variables in C

The following table shows C and C++ language declarations that are generated by the DB2 precompiler for some typical XML types. The declarations that the DB2 coprocessor generates might be different.

Table 51. Examples of C language variable declarations

You declare this variable	DB2 generates this variable
SQL TYPE IS XML AS BLOB (1M) blob_xml;	struct { unsigned long length; char data??(1048576??); } blob_xml;
SQL TYPE IS XML AS CLOB(40000K) clob_xml;	struct { unsigned long length; char data??(40960000??); } clob_xml;
SQL TYPE IS XML AS DBCLOB (4000K) dbclob_xml;	struct { unsigned long length; unsigned short data??(4096000??); } dbclob_xml;
SQL TYPE IS XML AS BLOB_FILE blob_xml_file;	struct { unsigned long name_length; unsigned long data_length; unsigned long file_options; char name??(255??); } blob_xml_file;
SQL TYPE IS XML AS CLOB_FILE clob_xml_file;	struct { unsigned long name_length; unsigned long data_length; unsigned long file_options; char name??(255??); } clob_xml_file;
SQL TYPE IS XML AS DBCLOB_FILE dbclob_xml_file;	struct { unsigned long name_length; unsigned long data_length; unsigned long file_options; char name??(255??); } dbclob_xml_file;

Declarations of XML host variables in COBOL

The declarations that are generated for COBOL differ, depending on whether you use the DB2 precompiler or the DB2 coprocessor.

The following table shows COBOL declarations that the DB2 precompiler generates for some typical XML types.

Table 52. Examples of COBOL variable declarations by the DB2 precompiler

You declare this variable	DB2 precompiler generates this variable
01 BLOB-XML USAGE IS SQL TYPE IS XML AS BLOB(1M).	01 BLOB-XML. 02 BLOB-XML-LENGTH PIC 9(9) COMP. 02 BLOB-XML-DATA. 49 FILLER PIC X(32767). ¹ 49 FILLER PIC X(32767). <i>Repeat 30 times</i> : 49 FILLER PIC X(1048576-32*32767).
01 CLOB-XML USAGE IS SQL TYPE IS XML AS CLOB(40000K).	01 CLOB-XML. 02 CLOB-XML-LENGTH PIC 9(9) COMP. 02 CLOB-XML-DATA. 49 FILLER PIC X(32767). ¹ 49 FILLER PIC X(32767). <i>Repeat 1248 times</i> : 49 FILLER PIC X(40960000-1250*32767).
01 DBCLOB-XML USAGE IS SQL TYPE IS XML AS DBCLOB(4000K).	01 DBCLOB-XML. 02 DBCLOB-XML-LENGTH PIC 9(9) COMP. 02 DBCLOB-XML-DATA. 49 FILLER PIC G(32767) USAGE DISPLAY-1. ² 49 FILLER PIC G(32767) USAGE DISPLAY-1. <i>Repeat 123 times</i> : 49 FILLER PIC G(4096000-125*32767) USAGE DISPLAY-1.
01 BLOB-XML-FILE USAGE IS SQL TYPE IS XML AS BLOB-FILE.	01 BLOB-XML-FILE. 49 BLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 BLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5. 49 BLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 BLOB-XML-FILE-NAME PIC X(255).
01 CLOB-XML-FILE USAGE IS SQL TYPE IS XML AS CLOB-FILE.	01 CLOB-XML-FILE. 49 CLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 CLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5. 49 CLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 CLOB-XML-FILE-NAME PIC X(255).
01 DBCLOB-XML-FILE USAGE IS SQL TYPE IS XML AS DBCLOB-FILE.	01 DBCLOB-XML-FILE. 49 DBCLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 DBCLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5. 49 DBCLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 DBCLOB-XML-FILE-NAME PIC X(255).

Notes:

1. For XML AS BLOB or XML AS CLOB host variables that are greater than 32767 bytes in length, DB2 creates multiple host language declarations of 32767 or fewer bytes.
2. For XML AS DBCLOB host variables that are greater than 32767 double-byte characters in length, DB2 creates multiple host language declarations of 32767 or fewer double-byte characters.

Declarations of XML host variables in PL/I

The declarations that are generated for PL/I differ, depending on whether you use the DB2 precompiler or the DB2 coprocessor.

The following table shows PL/I declarations that the DB2 precompiler generates for some typical XML types.




Table 53. Examples of PL/I variable declarations

You declare this variable	DB2 precompiler generates this variable
DCL BLOB_XML SQL TYPE IS XML AS BLOB (1M);	DCL 1 BLOB_XML, 2 BLOB_XML_LENGTH BIN FIXED(31), 2 BLOB_XML_DATA, ¹ 3 BLOB_XML_DATA1 (32) CHAR(32767), 3 BLOB_XML_DATA2 CHAR(32);
DCL CLOB_XML SQL TYPE IS XML AS CLOB (40000K);	DCL 1 CLOB_XML, 2 CLOB_XML_LENGTH BIN FIXED(31), 2 CLOB_XML_DATA, ¹ 3 CLOB_XML_DATA1 (1250) CHAR(32767), 3 CLOB_XML_DATA2 CHAR(1250);
DCL DBCLOB_XML SQL TYPE IS XML AS DBCLOB (4000K);	DCL 1 DBCLOB_XML, 2 DBCLOB_XML_LENGTH BIN FIXED(31), 2 DBCLOB_XML_DATA, ² 3 DBCLOB_XML_DATA1 (250) GRAPHIC(16383), 3 DBCLOB_XML_DATA2 GRAPHIC(250);
DCL BLOB_XML_FILE SQL TYPE IS XML AS BLOB_FILE;	DCL 1 BLOB_XML_FILE, 2 BLOB_XML_FILE_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 BLOB_XML_FILE_DATA_LENGTH BIN FIXED(31), 2 BLOB_XML_FILE_FILE_OPTIONS BIN FIXED(31), 2 BLOB_XML_FILE_NAME CHAR(255);
DCL CLOB_XML_FILE SQL TYPE IS XML AS CLOB_FILE;	DCL 1 CLOB_XML_FILE, 2 CLOB_XML_FILE_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 CLOB_XML_FILE_DATA_LENGTH BIN FIXED(31), 2 CLOB_XML_FILE_FILE_OPTIONS BIN FIXED(31), 2 CLOB_XML_FILE_NAME CHAR(255);
DCL DBCLOB_XML_FILE SQL TYPE IS XML AS DBCLOB_FILE;	DCL 1 DBCLOB_XML_FILE, 2 DBCLOB_XML_FILE_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 DBCLOB_XML_FILE_DATA_LENGTH BIN FIXED(31), 2 DBCLOB_XML_FILE_FILE_OPTIONS BIN FIXED(31), 2 DBCLOB_XML_FILE_NAME CHAR(255);

Table 53. Examples of PL/I variable declarations (continued)

You declare this variable	DB2 precompiler generates this variable
Notes:	
1. For XML AS BLOB or XML AS CLOB host variables that are greater than 32767 bytes in length, DB2 creates host language declarations in the following way: <ul style="list-style-type: none"> • If the length of the XML is greater than 32767 bytes and evenly divisible by 32767, DB2 creates an array of 32767-byte strings. The dimension of the array is $length/32767$. • If the length of the XML is greater than 32767 bytes but not evenly divisible by 32767, DB2 creates two declarations: The first is an array of 32767 byte strings, where the dimension of the array, n, is $length/32767$. The second is a character string of length $length-n*32767$. 	
2. For XML AS DBCLOB host variables that are greater than 16383 double-byte characters in length, DB2 creates host language declarations in the following way: <ul style="list-style-type: none"> • If the length of the XML is greater than 16383 characters and evenly divisible by 16383, DB2 creates an array of 16383-character strings. The dimension of the array is $length/16383$. • If the length of the XML is greater than 16383 characters but not evenly divisible by 16383, DB2 creates two declarations: The first is an array of 16383 byte strings, where the dimension of the array, m, is $length/16383$. The second is a character string of length $length-m*16383$. 	

Related concepts:

-  Insertion of rows with XML column values (DB2 Programming for XML)
-  Retrieving XML data (DB2 Programming for XML)
-  Updates of XML columns (DB2 Programming for XML)

XML column updates in embedded SQL applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the textual XML format.

The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the database server as character data is treated as externally encoded data.

Externally encoded data can have internal encoding. That is, the data might be sent to the database server as character data, but the data contains encoding information. DB2 does not enforce consistency of the internal and external encoding. When the internal and external encoding information differs, the external encoding takes precedence. However, if there is a difference between the external and internal encoding, intervening character conversion might have occurred on the data, and there might be data loss.

Character data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

The following examples demonstrate how to update XML columns in assembler, C, COBOL, and PL/I applications. The examples use a table named MYCUSTOMER, which is a copy of the sample CUSTOMER table.

Example: The following example shows an assembler program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server

honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```

*****
* UPDATE AN XML COLUMN WITH DATA IN AN XML AS CLOB HOST VARIABLE *
*****
EXEC SQL
    UPDATE MYCUSTOMER
    SET INFO = :XMLBUF
    WHERE CID = 1000
*****
* UPDATE AN XML COLUMN WITH DATA IN AN XML AS BLOB HOST VARIABLE *
*****
EXEC SQL
    UPDATE MYCUSTOMER
    SET INFO = :XMLBLOB
    WHERE CID = 1000
*****
* UPDATE AN XML COLUMN WITH DATA IN A CLOB HOST VARIABLE. USE *
* THE XMLPARSE FUNCTION TO CONVERT THE DATA TO THE XML TYPE. *
*****
EXEC SQL
    UPDATE MYCUSTOMER
    SET INFO = XMLPARSE(DOCUMENT :CLOBBUF)
    WHERE CID = 1000
...
LTORG
*****
* HOST VARIABLE DECLARATIONS *
*****
XMLBUF SQL TYPE IS XML AS CLOB 10K
XMLBLOB SQL TYPE IS XML AS BLOB 10K
CLOBBUF SQL TYPE IS CLOB 10K

```

Example: The following example shows a C language program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```

/*****/
/* Host variable declarations */
/*****/
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;
/*****/
/* Update an XML column with data in an XML AS CLOB host variable */
/*****/
EXEC SQL UPDATE MYCUSTOMER SET INFO = :xmlBuf where CID = 1000;
/*****/
/* Update an XML column with data in an XML AS BLOB host variable */
/*****/
EXEC SQL UPDATE MYCUSTOMER SET INFO = :xmlblob where CID = 1000;
/*****/
/* Update an XML column with data in a CLOB host variable. Use */
/* the XMLPARSE function to convert the data to the XML type. */
/*****/
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :clobBuf) where CID = 1000;

```

Example: The following example shows a COBOL program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors

the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```

*****
* Host variable declarations *
*****
  01 XMLBUF USAGE IS SQL TYPE IS XML AS CLOB(10K).
  01 XMLBLOB  USAGE IS SQL TYPE IS XML AS BLOB(10K).
  01 CLOBBUF  USAGE IS SQL TYPE IS CLOB(10K).
*****
* Update an XML column with data in an XML AS CLOB host variable *
*****
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBUF where CID = 1000.
*****
* Update an XML column with data in an XML AS BLOB host variable *
*****
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBLOB where CID = 1000.
*****
* Update an XML column with data in a CLOB host variable. Use      *
* the XMLPARSE function to convert the data to the XML type.      *
*****
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :CLOBBUF) where CID = 1000.

```


Example: The following example shows a PL/I program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```

/*****
/* Host variable declarations */
/*****
DCL
  XMLBUF SQL TYPE IS XML AS CLOB(10K),
  XMLBLOB SQL TYPE IS XML AS BLOB(10K),
  CLOBBUF SQL TYPE IS CLOB(10K);
/*****
/* Update an XML column with data in an XML AS CLOB host variable */
/*****
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBUF where CID = 1000;
/*****
/* Update an XML column with data in an XML AS BLOB host variable */
/*****
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBLOB where CID = 1000;
/*****
/* Update an XML column with data in a CLOB host variable. Use      */
/* the XMLPARSE function to convert the data to the XML type.      */
/*****
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :CLOBBUF) where CID = 1000;

```

 Insertion of rows with XML column values (DB2 Programming for XML)

 Updates of XML columns (DB2 Programming for XML)

XML data retrieval in embedded SQL applications

In an embedded SQL application, if you retrieve the data into a character host variable, DB2 converts the data from the UTF-8 encoding scheme to the application encoding scheme. If you retrieve the data into binary host variable, DB2 does not convert the data to another encoding scheme.

The output data is in the textual XML format.

DB2 might add an XML encoding specification to the retrieved data, depending on whether you call the XMLSERIALIZE function when you retrieve the data. If you

do not call the XMLSERIALIZE function, DB2 adds the correct XML encoding specification to the retrieved data. If you call the XMLSERIALIZE function, DB2 adds an internal XML encoding declaration for UTF-8 encoding if you specify INCLUDING XMLDECLARATION in the function call. When you use INCLUDING XMLDECLARATION, you need to ensure that the retrieved data is not converted from UTF-8 encoding to another encoding.

The following examples demonstrate how to retrieve data from XML columns in assembler, C, COBOL, and PL/I applications. The examples use a table named MYCUSTOMER, which is a copy of the sample CUSTOMER table.

Example: The following example shows an assembler program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```
*****
* RETRIEVE XML COLUMN DATA INTO AN XML AS CLOB HOST VARIABLE      *
*****
      EXEC SQL                                                    +
          SELECT INFO                                            +
          INTO :XMLBUF                                          +
          FROM MYCUSTOMER                                       +
          WHERE CID = 1000
*****
* RETRIEVE XML COLUMN DATA INTO AN XML AS BLOB HOST VARIABLE      *
*****
      EXEC SQL                                                    +
          SELECT INFO                                            +
          INTO :XMLBLOB                                         +
          FROM MYCUSTOMER                                       +
          WHERE CID = 1000
*****
* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.    *
* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE       *
* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML       *
* TYPE TO THE CLOB TYPE.                                         *
*****
      EXEC SQL                                                    +
          SELECT XMLSERIALIZE(INFO AS CLOB(10K))                +
          INTO :CLOBBUF                                         +
          FROM MYCUSTOMER                                       +
          WHERE CID = 1000
...
      LTORG
*****
* HOST VARIABLE DECLARATIONS *
*****
XMLBUF   SQL TYPE IS XML AS CLOB 10K
XMLBLOB  SQL TYPE IS XML AS BLOB 10K
CLOBBUF  SQL TYPE IS CLOB 10K
```

Example: The following example shows a C language program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with

UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```

/*****/
/* Host variable declarations */
/*****/
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
SQL TYPE IS XML AS BLOB( 10K ) xmlBlob;
SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;
/*****/
/* Retrieve data from an XML column into an XML AS CLOB host variable */
/*****/
EXEC SQL SELECT INFO INTO :xmlBuf from myTable where CID = 1000;
/*****/
/* Retrieve data from an XML column into an XML AS BLOB host variable */
/*****/
EXEC SQL SELECT INFO INTO :xmlBlob from myTable where CID = 1000;
/*****/
/* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE. */
/* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE */
/* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML */
/* TYPE TO THE CLOB TYPE. */
/*****/
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
INTO :clobBuf from myTable where CID = 1000;

```

Example: The following example shows a COBOL program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```

*****
* Host variable declarations *
*****
01 XMLBUF USAGE IS SQL TYPE IS XML AS CLOB(10K).
01 XMLBLOB USAGE IS SQL TYPE IS XML AS BLOB(10K).
01 CLOBBUF USAGE IS SQL TYPE IS CLOB(10K).
*****
* Retrieve data from an XML column into an XML AS CLOB host variable *
*****
EXEC SQL SELECT INFO
INTO :XMLBUF
FROM MYTABLE
WHERE CID = 1000
END-EXEC.
*****
* Retrieve data from an XML column into an XML AS BLOB host variable *
*****
EXEC SQL SELECT INFO
INTO :XMLBLOB
FROM MYTABLE
WHERE CID = 1000
END-EXEC.

```



```

*****
* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.      *
* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE          *
* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML          *
* TYPE TO THE CLOB TYPE.                                           *
*****
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
      INTO :CLOBBUF
      FROM MYTABLE
      WHERE CID = 1000
END-EXEC.

```

Example: The following example shows a PL/I program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```

/*****/
/* Host variable declarations */
/*****/
DCL
  XMLBUF SQL TYPE IS XML AS CLOB(10K),
  XMLBLOB SQL TYPE IS XML AS BLOB(10K),
  CLOBBUF SQL TYPE IS CLOB(10K);
/*****/
/* Retrieve data from an XML column into an XML AS CLOB host variable */
/*****/
EXEC SQL SELECT INFO  INTO :XMLBUF FROM MYTABLE WHERE CID = 1000;
/*****/
/* Retrieve data from an XML column into an XML AS BLOB host variable */
/*****/
EXEC SQL SELECT INFO  INTO :XMLBLOB FROM MYTABLE WHERE CID = 1000;
/*****/
/* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.      */
/* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE          */
/* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML          */
/* TYPE TO THE CLOB TYPE.                                           */
/*****/
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
      INTO :CLOBBUF FROM MYTABLE WHERE CID = 1000;

```

 Retrieving XML data (DB2 Programming for XML)

Programming examples

You can write DB2 programs in assembler language, C, C++, COBOL, Fortran, PL/I, or REXX. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements.

You can write DB2 programs in assembler language, C, C++, COBOL, Fortran, PL/I or REXX. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in DSN10.SDSNSAMP as a model for your JCL.

Conventions used in examples of coding SQL statements

The examples in this information use certain conventions and assumptions. Some of the examples vary from these conventions. Exceptions are noted where they occur.

The SQL statements in this information use the following conventions:

- The SQL statement is part of a C or COBOL application program. Each SQL example is displayed on several lines, with each clause of the statement on a separate line.
- The use of the precompiler options APOST and APOSTSQL are assumed (although they are not the defaults). Therefore, apostrophes (') are used to delimit character string literals within SQL and host language statements.
- The SQL statements access data in the sample tables provided with DB2. The tables contain data that a manufacturing company might keep about its employees and its current projects.
- An SQL example does not necessarily show the complete syntax of an SQL statement.
- Examples do not take referential constraints into account.

Related concepts:

“DB2 sample applications” on page 1092


“Programming examples in assembler” on page 271


“Programming examples in C” on page 311

“Programming examples in COBOL” on page 363

“Programming examples in PL/I” on page 432

“Programming examples in REXX” on page 448

 C and C++ language options to use with the installation verification procedures (DB2 Installation and Migration)

 COBOL options to use with the installation verification procedures (DB2 Installation and Migration)

 PL/I options to use with the installation verification procedures (DB2 Installation and Migration)

Related reference:

 DB2 sample tables (Introduction to DB2 for z/OS)

Examples of programs that call stored procedures

Examples can be used as models when you write applications that call stored procedures. In addition, DSN10.SDSNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

Chapter 4. Coding SQL statements in assembler application programs

When you code SQL statements in assembler application programs, you should follow certain guidelines.

Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler

Assembler programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

To define the SQL communications area, SQLSTATE, and SQLCODE:

Choose one of the following actions:

Option	Description
To define the SQL communications area:	<p>1. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration:</p> <pre>EXEC SQL INCLUDE SQLCA</pre> <p>If your program is reentrant, you must include the SQLCA within a unique data area that is acquired for your task (a DSECT). For example, at the beginning of your program, specify the following code:</p> <pre>PROGAREA DSECT EXEC SQL INCLUDE SQLCA</pre> <p>As an alternative, you can create a separate storage area for the SQLCA and provide addressability to that area.</p> <p>DB2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.</p>

Option	Description
<p>To declare <code>SQLCODE</code> and <code>SQLSTATE</code> host variables:</p>	<ol style="list-style-type: none"> 1. Declare the <code>SQLCODE</code> variable within a <code>BEGIN DECLARE SECTION</code> statement and an <code>END DECLARE SECTION</code> statement in your program declarations as a fullword integer. 2. Declare the <code>SQLSTATE</code> variable within a <code>BEGIN DECLARE SECTION</code> statement and an <code>END DECLARE SECTION</code> statement in your program declarations as a character string of length 5 (CL5). <p>Restriction: Do not declare an <code>SQLSTATE</code> variable as an element of a structure.</p> <p>Requirement: After you declare the <code>SQLCODE</code> and <code>SQLSTATE</code> variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</p>

Related tasks:

“Checking the execution of SQL statements” on page 227

“Checking the execution of SQL statements by using the `SQLCA`” on page 228

“Checking the execution of SQL statements by using `SQLCODE` and `SQLSTATE`” on page 232

“Defining the items that your program can use to check whether an SQL statement executed successfully” on page 173

Defining SQL descriptor areas in assembler

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

To define SQL descriptor areas:

Code the SQLDA directly in the program, or use the following SQL INCLUDE statement to request a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA
```

Restriction: You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the `TWOPASS` SQL processing option.

Related tasks:

“Defining SQL descriptor areas” on page 173

Declaring host variables and indicator variables in assembler

You can use host variables, host variable arrays, and host structures in SQL statements in your program to pass data between DB2 and your application.

To declare host variables, host variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:

- You can declare host variables in normal assembler style (DC or DS), depending on the data type and the limitations on that data type. You can specify a value on DC or DS declarations (for example, DC H'5'). The DB2 precompiler examines only packed decimal declarations.
 - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
 - If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
 - Ensure that any SQL statement that uses a host variable or host variable array is within the scope of the statement that declares that variable or array.
 - If you are using the DB2 precompiler, ensure that the names of host variables and host variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.
2. Optional: Define any associated indicator variables, arrays, and structures.

Related tasks:

“Declaring host variables and indicator variables” on page 174

Host variables in assembler

In assembler programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

Restrictions:

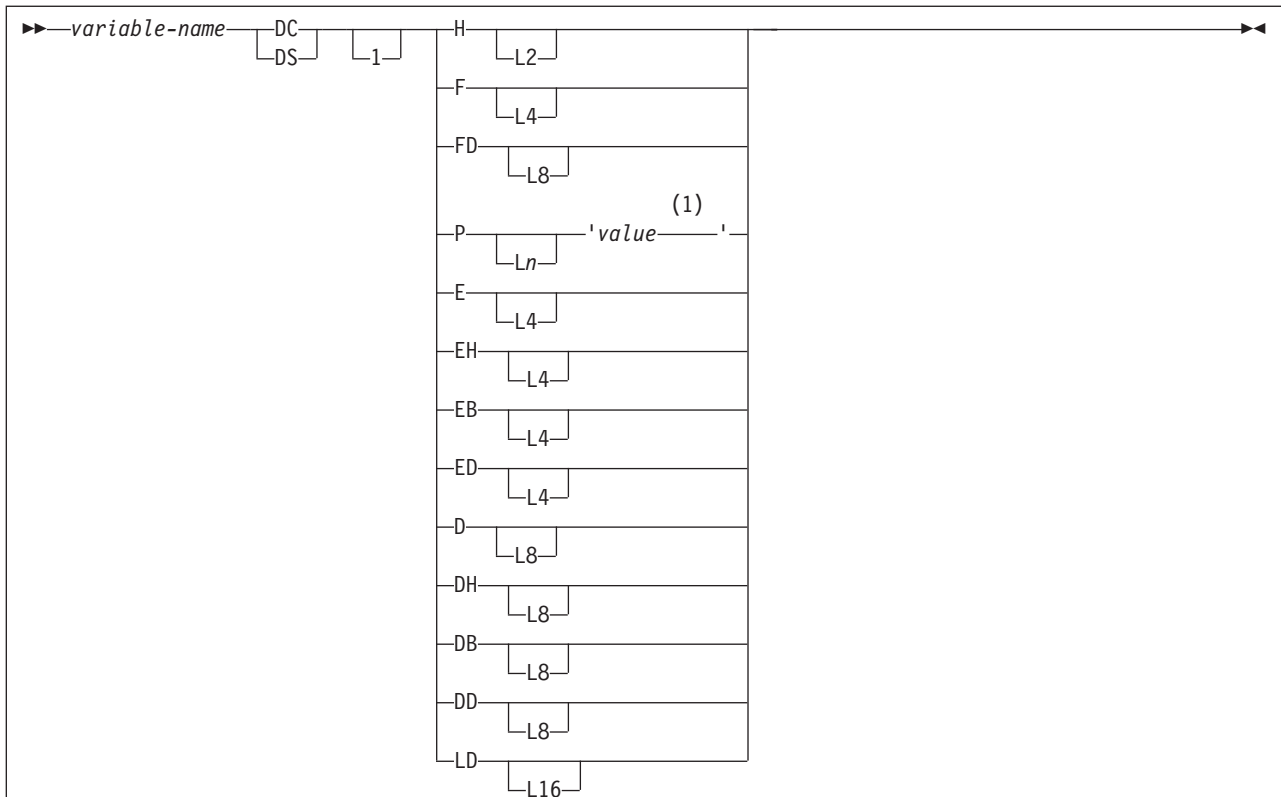
- Only some of the valid assembler declarations are valid host variable declarations. If the declaration for a host variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- The locator data types are assembler language data types and SQL data types. You cannot use locators as column types.

Recommendations:

- Be careful of overflow. For example, suppose that you retrieve an INTEGER column value into a DS H host variable, and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provide an indicator variable.
- Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a host variable that is declared as DS CL70, the rightmost ten characters of the retrieved string are truncated. If you retrieve a floating-point or decimal column value into a host variable declared as DS F, any fractional part of the value is removed.

Numeric host variables

The following diagram shows the syntax for declaring numeric host variables.



Notes:

- 1 *value* is a numeric value that specifies the scale of the packed decimal variable. If *value* does not include a decimal point, the scale is 0.

For floating-point data types (E, EH, EB, D, DH, and DB), use the FLOAT SQL processing option to specify whether the host variable is in IEEE binary floating-point or z/Architecture[®] hexadecimal floating-point format. If you specify FLOAT(S390), you need to define your floating-point host variables as E, EH, D, or DH. If you specify FLOAT(IEEE), you need to define your floating-point host variables as EB or DB. DB2 does not check if the host variable declarations or format of the host variable contents match the format that you specified with the FLOAT SQL processing option. Therefore, you need to ensure that your floating-point host variable types and contents match the format that you specified with the FLOAT SQL processing option. DB2 converts all floating-point input data to z/Architecture hexadecimal floating-point format before storing it.

Restriction: The FLOAT SQL processing options do not apply to the decimal floating-point host variable types ED, DD, or LD.

For the decimal floating-point host variable types ED, DD, and LD, you can specify the following special values: MIN, MAX, NAN, SNAN, and INFINITY.

Character host variables

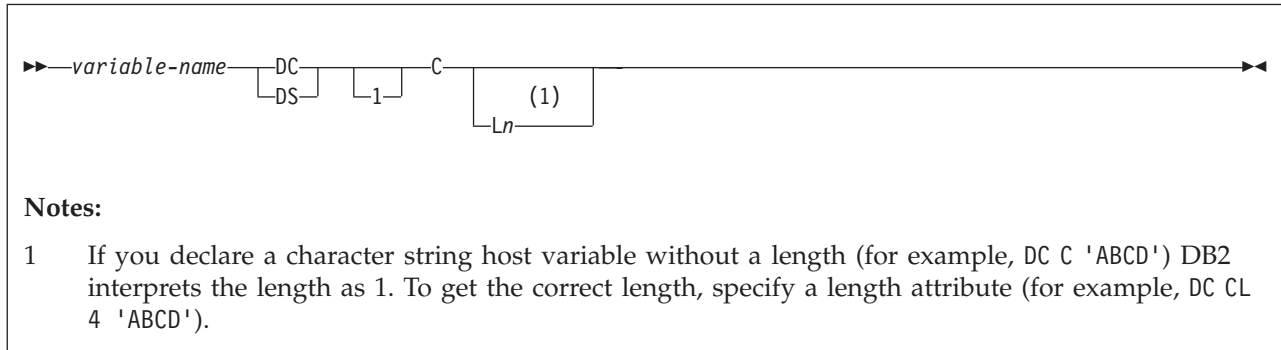
You can specify the following forms of character host variables:

- Fixed-length strings
- Varying-length strings

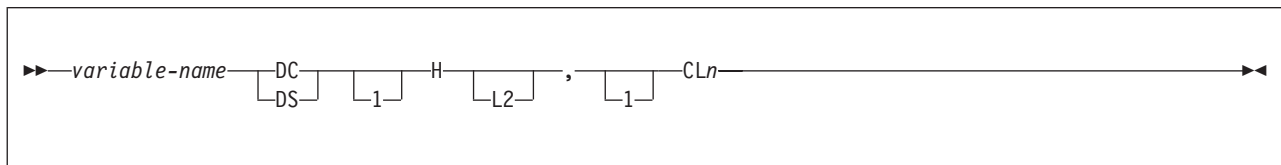
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring fixed-length character strings.



The following diagram shows the syntax for declaring varying-length character strings.



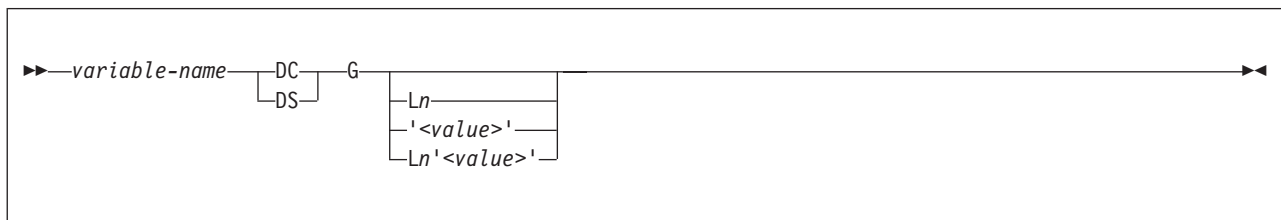
Graphic host variables

You can specify the following forms of graphic host variables:

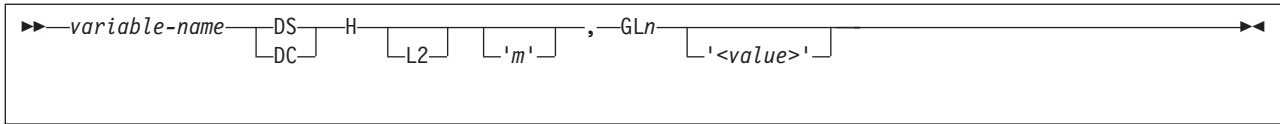
- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following diagrams show the syntax for forms other than DBCLOBs. In the syntax diagrams, *value* denotes one or more DBCS characters, and the symbols < and > represent the shift-out and shift-in characters.

The following diagram shows the syntax for declaring fixed-length graphic strings.

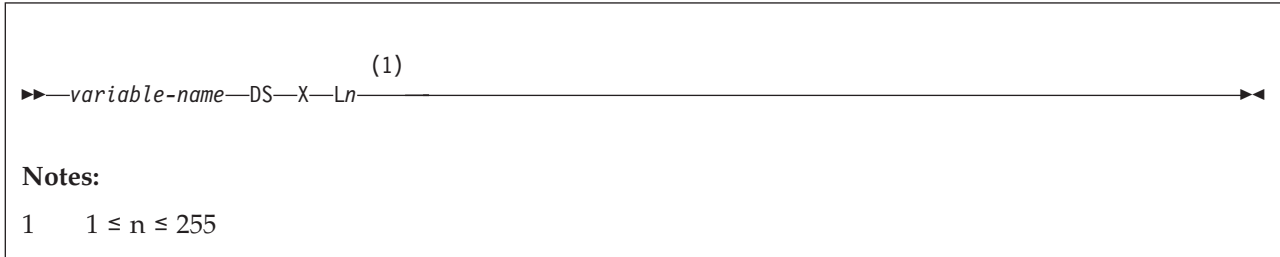


The following diagram shows the syntax for declaring varying-length graphic strings.



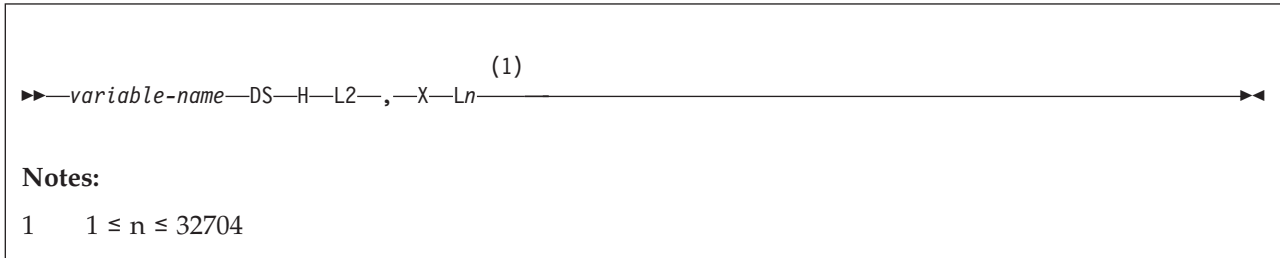
Binary host variables

The following diagram shows the syntax for declaring binary host variables.



Varbinary host variables

The following diagram shows the syntax for declaring varbinary host variables.



Result set locators

The following diagram shows the syntax for declaring result set locators.

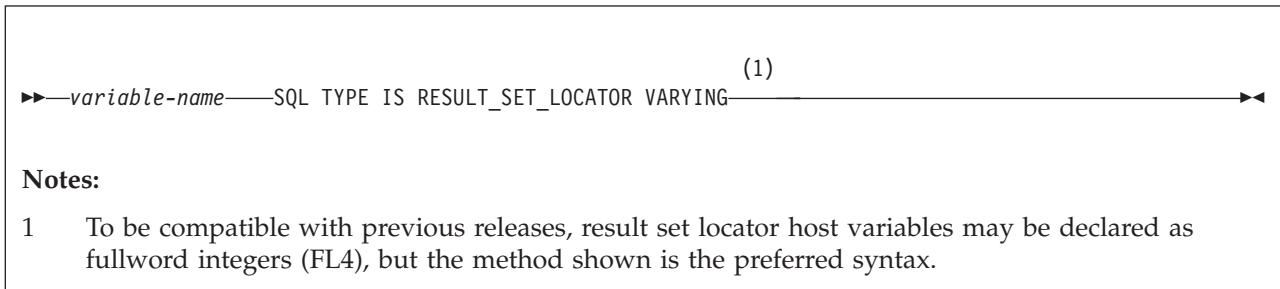


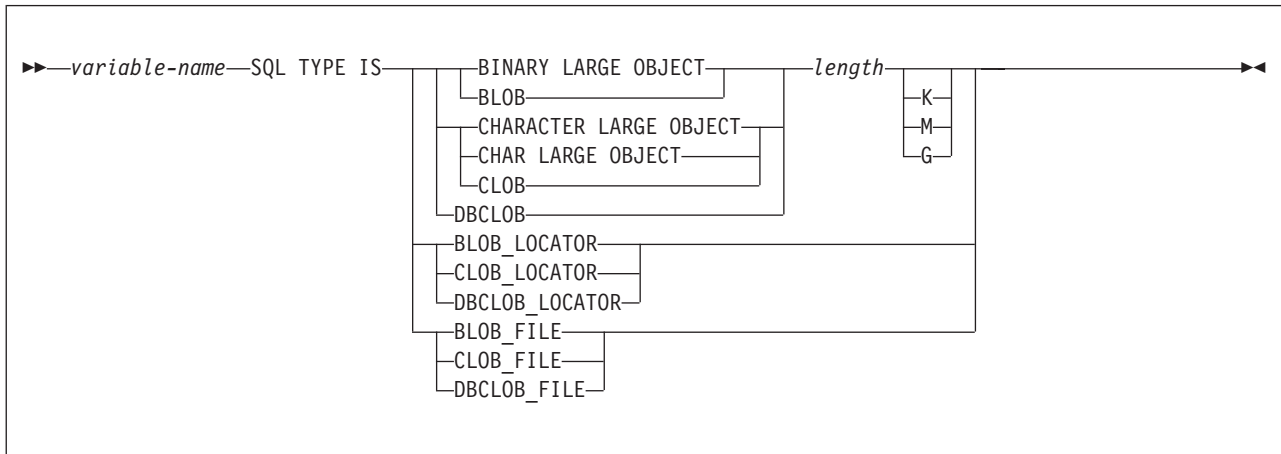
Table Locators

The following diagram shows the syntax for declaring of table locators.

▶▶ *variable-name*—SQL TYPE IS—TABLE LIKE—*table-name*—AS LOCATOR—▶▶

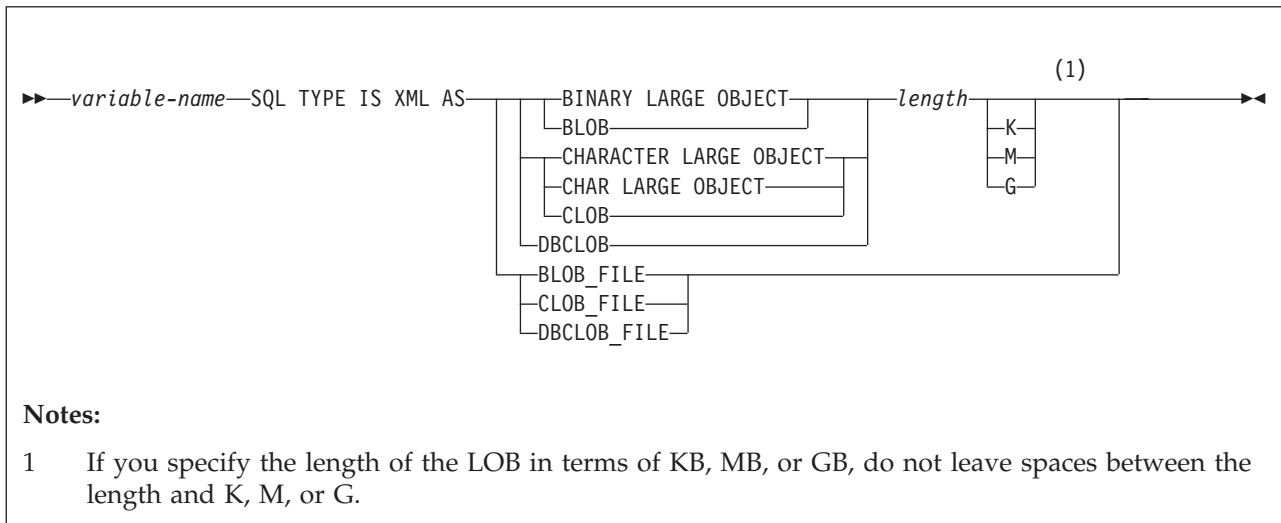
LOB variables, locators, and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables, locators, and file reference variables.



XML data host and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.



ROWIDs

The following diagram shows the syntax for declaring ROWID host variables.

▶▶ *variable-name*—SQL TYPE IS—ROWID—▶▶

Related concepts:

“Host variables” on page 174

“Rules for host variables in an SQL statement” on page 183

“Large objects (LOBs)” on page 465

Related tasks:

“Determining whether a retrieved value in a host variable is null or truncated” on page 186

“Inserting a single row by using a host variable” on page 189

“Inserting null values into columns by using indicator variables or arrays” on page 190

“Retrieving a single row of data into host variables” on page 184

“Updating data by using host variables” on page 189

Related reference:

“Descriptions of SQL processing options” on page 932

📖 High Level Assembler (HLASM) and Toolkit Feature Library

Indicator variables in assembler

An indicator variable is a 2-byte integer (DS HL2). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in assembler.

▶▶ *variable-name* — DC — DS — 1 — H — L2 —▶▶

Example

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,           X
                                :DAY :DAYIND,      X
                                :BGN :BGNIND,      X
                                :END :ENDIND
```

You can declare these variables as follows:

```
CLSCD   DS CL7
DAY     DS HL2
BGN     DS CL8
END     DS CL8
DAYIND  DS HL2           INDICATOR VARIABLE FOR DAY
BGNIND  DS HL2           INDICATOR VARIABLE FOR BGN
ENDIND  DS HL2           INDICATOR VARIABLE FOR END
```

Related concepts:

“Indicator variables, arrays, and structures” on page 176

Related tasks:

“Inserting null values into columns by using indicator variables or arrays” on page 190

Equivalent SQL and assembler data types

When you declare host variables in your assembler programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base `SQLTYPE` and `SQLLEN` values that the precompiler uses for host variables in SQL statements.

Table 54. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in assembler programs

Assembler host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
DS HL2	500	2	SMALLINT
DS FL4	496	4	INTEGER
DS P'value' DS PLn'value' or DS PLn 1<=n<=16	484	p in byte 1, s in byte 2	DECIMAL(p, s)
short decimal FLOAT: SDFP DC ED SDFP DC EDL4 SDFP DC EDL4'11.11'	996	4	DECFLOAT
long decimal FLOAT: LDFP DC DD LDFP DC DDL8 LDFP DC DDL8'22.22'	996	8	DECFLOAT
extended decimal FLOAT: EDFP DC LD EDFP DC LDL16 EDFP DC LDL16'33.33'	996	16	DECFLOAT
DS EL4 DS EHL4 DS EBL4	480	4	REAL or FLOAT (n) 1<=n<=21
DS DL8 DS DHL8 DS DBL8	480	8	DOUBLE PRECISION, or FLOAT (n) 22<=n<=53
DS FDL8 DS FD	492	8	BIGINT
SQL TYPE IS BINARY(n) 1<=n<=255	912	n	BINARY(n)
SQL TYPE IS VARBINARY(n) or SQL TYPE IS BINARY(n) VARYING 1<=n<=32704	908	n	VARBINARY(n)

Table 54. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in assembler programs (continued)

Assembler host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
DS CLn 1<=n<=255	452	n	CHAR(n)
DS HL2,CLn 1<=n<=255	448	n	VARCHAR(n)
DS HL2,CLn n>255	456	n	VARCHAR(n)
DS GLm 2<=m<=254	468	n	GRAPHIC(n) 3
2			
DS HL2,GLm 2<=m<=254	464	n	VARGRAPHIC(n) 3
2			
DS HL2,GLm m>254	472	n	VARGRAPHIC(n) 3
2			
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator ^{4,5}
SQL TYPE IS TABLE LIKE table-name AS LOCATOR	976	4	Table locator ⁴
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator ⁴
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator ⁴
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator ⁴
SQL TYPE IS BLOB(n) 1<=n<=2147483647	404	n	BLOB(n)
SQL TYPE IS CLOB(n) 1<=n<=2147483647	408	n	CLOB(n)
SQL TYPE IS DBCLOB(n) 1<=n<=1073741823	412	n	DBCLOB(n) 3
SQL TYPE IS XML AS BLOB(n)	404	0	XML
SQL TYPE IS XML AS CLOB(n)	408	0	XML
SQL TYPE IS XML AS DBCLOB(n)	412	0	XML
SQL TYPE IS BLOB_FILE	916/917	267	BLOB file reference ⁴
SQL TYPE IS CLOB_FILE	920/921	267	CLOB file reference ⁴

Table 54. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in assembler programs (continued)

Assembler host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
SQL TYPE IS DBCLOB_FILE	924/925	267	DBCLOB file reference ⁴
SQL TYPE IS XML AS BLOB_FILE	916/917	267	XML BLOB file reference ⁴
SQL TYPE IS XML AS CLOB_FILE	920/921	267	XML CLOB file reference ⁴
SQL TYPE IS XML AS DBCLOB_FILE	924/925	267	XML DBCLOB file reference ⁴
SQL TYPE IS ROWID	904	40	ROWIDnote 5

Notes:

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
2. *m* is the number of bytes.
3. *n* is the number of double-byte characters.
4. This data type cannot be used as a column type.
5. To be compatible with previous releases, result set locator host variables may be declared as fullword integers (FL4), but the method shown is the preferred syntax.

The following table shows equivalent assembler host variables for each SQL data type. Use this table to determine the assembler data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define variable DS CL*n*.

This table shows direct conversions between SQL data types and assembler data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 converts those compatible data types.

Table 55. Assembler host variable equivalents that you can use when retrieving data of a particular SQL data type

SQL data type	Assembler host variable equivalent	Notes
SMALLINT	DS HL2	
INTEGER	DS F	
BIGINT	DS FD OR DS FDL8	DS FDL8 requires High Level Assembler (HLASM), Release 4 or later.

Table 55. Assembler host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	Assembler host variable equivalent	Notes
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	DS P' <i>value</i> ' DS PL <i>n</i> ' <i>value</i> ' DS PL <i>n</i>	<i>p</i> is precision; <i>s</i> is scale. 1<= <i>p</i> <=31 and 0<= <i>s</i> <= <i>p</i> . 1<= <i>n</i> <=16. <i>value</i> is a literal value that includes a decimal point. You must use <i>Ln</i> , <i>value</i> , or both. Using only <i>value</i> is recommended. Precision: If you use <i>Ln</i> , it is 2 <i>n</i> -1; otherwise, it is the number of digits in <i>value</i> . Scale: If you use <i>value</i> , it is the number of digits to the right of the decimal point; otherwise, it is 0. For efficient use of indexes: Use <i>value</i> . If <i>p</i> is <i>even</i> , do not use <i>Ln</i> and be sure the precision of <i>value</i> is <i>p</i> and the scale of <i>value</i> is <i>s</i> . If <i>p</i> is <i>odd</i> , you can use <i>Ln</i> (although it is not advised), but you must choose <i>n</i> so that 2 <i>n</i> -1= <i>p</i> , and <i>value</i> so that the scale is <i>s</i> . Include a decimal point in <i>value</i> , even when the scale of <i>value</i> is 0.
REAL or FLOAT(<i>n</i>)	DS EL4 DS EHL4 DS EBL4 ¹	1<= <i>n</i> <=21
DOUBLE PRECISION, DOUBLE, or FLOAT(<i>n</i>)	DS DL8 DS DHL8 DS DBL8 ¹	22<= <i>n</i> <=53
DECFLOAT	DC EDL4 DC DDL8 DC LDL16	
CHAR(<i>n</i>)	DS CL <i>n</i>	1<= <i>n</i> <=255
VARCHAR(<i>n</i>)	DS HL2,CL <i>n</i>	
GRAPHIC(<i>n</i>)	DS GL <i>m</i>	<i>m</i> is expressed in bytes. <i>n</i> is the number of double-byte characters. 1<= <i>n</i> <=127
VARGRAPHIC(<i>n</i>)	DS HL2,GL <i>x</i> DS HL2' <i>m</i> ',GL <i>x</i> '< <i>value</i> >'	<i>x</i> and <i>m</i> are expressed in bytes. <i>n</i> is the number of double-byte characters. < and > represent shift-out and shift-in characters.
BINARY(<i>n</i>)	Format 1: variable-name-- DS--X--Ln Format 2: SQL TYPE IS BINARY(<i>n</i>)	1<= <i>n</i> <=255

Table 55. Assembler host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	Assembler host variable equivalent	Notes
VARBINARY(<i>n</i>)	<p>Format 1: variable-name-- DS--H--L2-- ,-- X--Ln</p> <p>Format 2: SQL TYPE IS VARBINARY(<i>n</i>) or SQL TYPE IS BINARY(<i>n</i>) VARYING</p>	1 <= <i>n</i> <= 32704
DATE	DS CL <i>n</i>	If you are using a date exit routine, <i>n</i> is determined by that routine; otherwise, <i>n</i> must be at least 10.
TIME	DS CL <i>n</i>	If you are using a time exit routine, <i>n</i> is determined by that routine. Otherwise, <i>n</i> must be at least 6; to include seconds, <i>n</i> must be at least 8.
TIMESTAMP	DS CL <i>n</i>	<i>n</i> must be at least 19. To include microseconds, <i>n</i> must be 26; if <i>n</i> is less than 26, truncation occurs on the microseconds part.
TIMESTAMP(0)	DS CL <i>n</i>	<i>n</i> must be at least 19.
TIMESTAMP(<i>p</i>) <i>p</i> > 0	DS CL <i>n</i>	<i>n</i> must be at least 19. To include fractional seconds, <i>n</i> must be 20+ <i>x</i> where <i>x</i> is the number of fractional seconds to include; if <i>x</i> is less than <i>p</i> , truncation occurs on the fractional seconds part.
TIMESTAMP(0) WITH TIME ZONE	DS HL2,CL <i>n</i>	<i>n</i> must be at least 25.
TIMESTAMP(<i>p</i>) WITH TIME ZONE <i>p</i> > 0	DS HL2,CL <i>n</i>	<i>n</i> must be at least 26+ <i>p</i> .
Result set locator	DS F	Use this data type only to receive result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.

Table 55. Assembler host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	Assembler host variable equivalent	Notes
BLOB(<i>n</i>)	SQL TYPE IS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
CLOB(<i>n</i>)	SQL TYPE IS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
DBCLOB(<i>n</i>)	SQL TYPE IS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823
XML	SQL TYPE IS XML AS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823
BLOB file reference	SQL TYPE IS BLOB_FILE	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB file reference	SQL TYPE IS CLOB_FILE	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB file reference	SQL TYPE IS DBCLOB_FILE	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
XML BLOB file reference	SQL TYPE IS XML AS BLOB_FILE	Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.
XML CLOB file reference	SQL TYPE IS XML AS CLOB_FILE	Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.
XML DBCLOB file reference	SQL TYPE IS XML AS DBCLOB_FILE	Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.
ROWID	SQL TYPE IS ROWID	

Notes:

1. Although stored procedures and user-defined functions can use IEEE floating-point host variables, you cannot declare a user-defined function or stored procedure parameter as IEEE.

Related concepts:

“Compatibility of SQL and language data types” on page 180

“LOB host variable, LOB locator, and LOB file reference variable declarations” on page 757

“Host variable data types for XML data in embedded SQL applications” on page 241

SQL statements in assembler programs

You can code SQL statements in an assembler program wherever you can use executable statements.

Each SQL statement in an assembler program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in an assembler program as follows:

```
EXEC SQL UPDATE DSN8A10.DEPT           X
          SET MGRNO = :MGRNUM          X
          WHERE DEPTNO = :INTDEPT
```

Comments: You cannot include assembler comments in SQL statements. However, you can include SQL comments in any embedded SQL statement.

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for assembler statements, except that you must specify EXEC SQL within one line. Any part of the statement that does not fit on one line can appear on subsequent lines, beginning at the continuation margin (column 16, the default). Every line of the statement, except the last, must have a continuation character (a non-blank character) immediately after the right margin in column 72.

Declaring tables and views: Your assembler program should include a DECLARE statement to describe each table and view the program accesses.

Including code: To include SQL statements or assembler host variable declaration statements from a member of a partitioned data set, place the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements.

Margins: Use the precompiler option MARGINS to set a left margin, a right margin, and a continuation margin. The default values for these margins are columns 1, 71, and 16, respectively. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement. If you use the default margins, you can place an SQL statement anywhere between columns 2 and 71.

Multiple-row FETCH statements: You can use only the FETCH ... USING DESCRIPTOR form of the multiple-row FETCH statement in an assembler program. The DB2 precompiler does not recognize declarations of host variable arrays for an assembler program.

Names: You can use any valid assembler name for a host variable. However, do not use external entry names or access plan names that begin with 'DSN' or host variable names that begin with 'SQL'. These names are reserved for DB2.

The first character of a host variable that is used in embedded SQL cannot be an underscore. However, you can use an underscore as the first character in a symbol that is **not** used in embedded SQL.

Statement labels: You can prefix an SQL statement with a label. The first line of an SQL statement can use a label beginning in the left margin (column 1). If you do not use a label, leave column 1 blank.

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be a label in the assembler source code and must be within the scope of the SQL statements that WHENEVER affects.

Special assembler considerations: The following considerations apply to programs written in assembler:

- To allow for reentrant programs, the precompiler puts all the variables and structures it generates within a DSECT called SQLDSECT, and it generates an assembler symbol called SQLDLLEN. SQLDLLEN contains the length of the DSECT. Your program must allocate an area of the size indicated by SQLDLLEN, initialize it, and provide addressability to it as the DSECT SQLDSECT. The precompiler does not generate code to allocate the storage for SQLDSECT; the application program must allocate the storage.

CICS: An example of code to support reentrant programs, running under CICS, follows:

```
DFHEISTG DSECT
          DFHEISTG
          EXEC SQL INCLUDE SQLCA

*
          DS    0F
SQDWSREG EQU    R7
SQDWSTOR DS    (SQLDLLEN)C  RESERVE STORAGE TO BE USED FOR SQLDSECT

:
:

XXPROGRM DFHEIENT CODEREG=R12,EIBREG=R11,DATAREG=R13
*
*
*  SQL WORKING STORAGE
          LA    SQDWSREG,SQDWSTOR    GET ADDRESS OF SQLDSECT
          USING SQLDSECT,SQDWSREG    AND TELL ASSEMBLER ABOUT IT
*
```

In this example, the actual storage allocation is done by the DFHEIENT macro.

TSO: The sample program in *prefix.SDSNSAMP(DSNTIAD)* contains an example of how to acquire storage for the SQLDSECT in a program that runs in a TSO environment. The following example code contains pieces from *prefix.SDSNSAMP(DSNTIAD)* with explanations in the comments.

```
DSNTIAD  CSECT          CONTROL SECTION NAME
          SAVE  (14,12)  ANY SAVE SEQUENCE
          LR   R12,R15   CODE ADDRESSABILITY
          USING DSNTIAD,R12  TELL THE ASSEMBLER
          LR   R7,R1     SAVE THE PARM POINTER

*
* Allocate storage of size PRGSIZ1+SQLDSIZ, where:
* - PRGSIZ1 is the size of the DSNTIAD program area
* - SQLDSIZ is the size of the SQLDSECT, and declared
*   when the DB2 precompiler includes the SQLDSECT
*
          L    R6,PRGSIZ1  GET SPACE FOR USER PROGRAM
          A    R6,SQLDSIZ  GET SPACE FOR SQLDSECT
          GETMAIN R,LV=(6)  GET STORAGE FOR PROGRAM VARIABLES
          LR   R10,R1     POINT TO IT

*
* Initialize the storage
*
          LR   R2,R10     POINT TO THE FIELD
          LR   R3,R6      GET ITS LENGTH
          SR   R4,R4      CLEAR THE INPUT ADDRESS
          SR   R5,R5      CLEAR THE INPUT LENGTH
          MVCL R2,R4      CLEAR OUT THE FIELD

*
* Map the storage for DSNTIAD program area
*
          ST   R13,FOUR(R10)  CHAIN THE SAVEAREA PTRS
          ST   R10,EIGHT(R13) CHAIN SAVEAREA FORWARD
```

```

                LR    R13,R10          POINT TO THE SAVEAREA
                USING PRGAREA1,R13     SET ADDRESSABILITY
*
* Map the storage for the SQLDSECT
*
                LR    R9,R13          POINT TO THE PROGAREA
                A     R9,PRGSIZ1      THEN PAST TO THE SQLDSECT
                USING SQLDSECT,R9     SET ADDRESSABILITY
...
                LTORG
*****
*
*   DECLARE VARIABLES, WORK AREAS
*
*****
PRGAREA1 DSECT                                WORKING STORAGE FOR THE PROGRAM
...
                DS    0D
PRGSIZE1 EQU    *-PRGAREA1                    DYNAMIC WORKAREA SIZE
...
DSNTIAD CSECT                                RETURN TO CSECT FOR CONSTANT
PRGSIZ1 DC    A(PRGSIZE1)                    SIZE OF PROGRAM WORKING STORAGE
CA      DSECT
                EXEC SQL INCLUDE SQLCA
...

```

- DB2 does not process set symbols in SQL statements.
- Generated code can include more than two continuations per comment.
- Generated code uses literal constants (for example, =F'84'), so an LTORG statement might be necessary.
- Generated code uses registers 0, 1, 14, and 15. Register 13 points to a save area that the called program uses. Register 15 does not contain a return code after a call that is generated by an SQL statement.

CICS: A CICS application program uses the DFHEIENT macro to generate the entry point code. When using this macro, consider the following:

- If you use the default DATAREG in the DFHEIENT macro, register 13 points to the save area.
- If you use any other DATAREG in the DFHEIENT macro, you must provide addressability to a save area.

For example, to use SAVED, you can code instructions to save, load, and restore register 13 around each SQL statement as in the following example.

```

ST    13,SAVER13    SAVE REGISTER 13
LA    13,SAVED     POINT TO SAVE AREA
EXEC  SQL . . .
L     13,SAVER13    RESTORE REGISTER 13

```

- If you have an addressability error in precompiler-generated code because of input or output host variables in an SQL statement, check to make sure that you have enough base registers.
- Do not put CICS translator options in the assembly source code. Instead, pass the options to the translator by using the PARM field.

Handling SQL error return codes in assembler

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information about the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 229.

You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234.

DSNTIAR syntax:

```
CALL DSNTIAR,(sqlca, message, lrecl),MF=(E,PARM)
```

The DSNTIAR parameters have the following meanings:

sqlca

An SQL communication area.

message

An output area, defined as a varying-length string, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
LINES    EQU    10
LRECL    EQU    132

:
MSGGLRECL DC    AL4(LRECL)
MESSAGE   DS    H,CL(LINES*LRECL)
           ORG    MESSAGE
MESSAGEL DC    AL2(LINES*LRECL)
MESSAGE1  DS    CL(LRECL)      text line 1
MESSAGE2  DS    CL(LRECL)      text line 2

:
MESSAGEn DS    CL(LRECL)      text line n

:
CALL DSNTIAR, (SQLCA,MESSAGE,MSGGLRECL),MF=(E,PARM)
```

where MESSAGE is the name of the message output area, LINES is the number of lines in the message output area, and LRECL is the length of each line.

lrecl

A fullword containing the logical record length of output messages, between 72 and 240.

The expression MF=(E,PARM) is an z/OS macro parameter that indicates dynamic execution. PARM is the name of a data area that contains a list of pointers to the call parameters of DSNTIAR.

See “DB2 sample applications” on page 1092 for instructions on how to access and print the source code for the sample program.

CICS: If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL DSNTIAC,(eib,commarea,sqlca,msg,lrecl),MF=(E,PARM)
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

eib EXEC interface block

commarea
communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see member DSN8FRDO in the data set *prefix.SDSNSAMP*.

The assembler source code for DSNTIAC and job DSNTJ5A, which assembles and link-edits DSNTIAC, are also in the data set *prefix.SDSNSAMP*.

Related tasks:

“Including dynamic SQL in your program” on page 193

“Embedding SQL statements in your application” on page 183

“Handling SQL error codes” on page 239

 Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Delimiters in SQL statements in assembler programs

You must delimit SQL statements in your assembler program so that DB2 knows when a particular SQL statement ends.

Delimit an SQL statement in your assembler program with the beginning keyword EXEC SQL and an end of line or end of last continued line.

Macros for assembler applications

Data set DSNA10.SDSNMACS contains all DB2 macros that are available for use.

Programming examples in assembler

You can write DB2 programs in assembler. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in DSN910.SDSNSAMP as a model for your JCL.

Related reference:

“Programming examples” on page 251

Chapter 5. Coding SQL statements in C application programs

When you code SQL statements in C or C++ application programs, you should follow certain guidelines.

Defining the SQL communications area, SQLSTATE, and SQLCODE in C

C and C++ programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

To define the SQL communications area, SQLSTATE, and SQLCODE:

Choose one of the following actions:

Option	Description
To define the SQL communications area:	<ol style="list-style-type: none">Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration: <pre>EXEC SQL INCLUDE SQLCA</pre><p>The standard declaration includes both a structure definition and a static data area named 'sqlca'.</p><p>DB2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.</p>

Option	Description
<p>To declare SQLCODE and SQLSTATE host variables:</p>	<ol style="list-style-type: none"> <li data-bbox="933 220 1419 409">1. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as a long integer: <pre>long SQLCODE;</pre> <li data-bbox="933 409 1419 598">2. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as a character array of length 6: <pre>char SQLSTATE[6];</pre> <p>Restriction: Do not declare an SQLSTATE variable as an element of a structure. Requirement: After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</p>

Related tasks:

“Checking the execution of SQL statements” on page 227

“Checking the execution of SQL statements by using the SQLCA” on page 228

“Checking the execution of SQL statements by using SQLCODE and SQLSTATE” on page 232

“Defining the items that your program can use to check whether an SQL statement executed successfully” on page 173

Defining SQL descriptor areas in C

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

To define SQL descriptor areas:

Code the SQLDA directly in the program, or use the following SQL INCLUDE statement to request a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA
```

You can place an SQLDA declaration wherever C allows a structure definition. Normal C scoping rules apply. The standard declaration includes only a structure definition with the name sqlda.

Restriction: You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.

Related tasks:

“Defining SQL descriptor areas” on page 173

Declaring host variables and indicator variables in C

You can use host variables, host variable arrays, and host structures in SQL statements in your program to pass data between DB2 and your application.

To declare host variables, host variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:
 - You can have more than one host variable declaration section in your program.
 - You can use class members as host variables. Class members that are used as host variables are accessible to any SQL statement within the class. However, you cannot use class objects as host variables.
 - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.

Restriction: The DB2 coprocessor for C/C++ supports only the ONEPASS option.

- If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
 - Ensure that any SQL statement that uses a host variable or host variable array is within the scope of the statement that declares that variable or array.
 - If you are using the DB2 precompiler, ensure that the names of host variables and host variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.
2. Optional: Define any associated indicator variables, arrays, and structures.

Related tasks:

“Declaring host variables and indicator variables” on page 174

Host variables in C

In C and C++ programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

Restrictions:

- Only some of the valid C declarations are valid host variable declarations. If the declaration for a variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- C supports some data types and storage classes with no SQL equivalents, such as register storage class, typedef, and long long.
- The following locator data types are special SQL data types that do not have C equivalents:
 - Result set locator
 - Table locator

- LOB locators

You cannot use them to define column types.

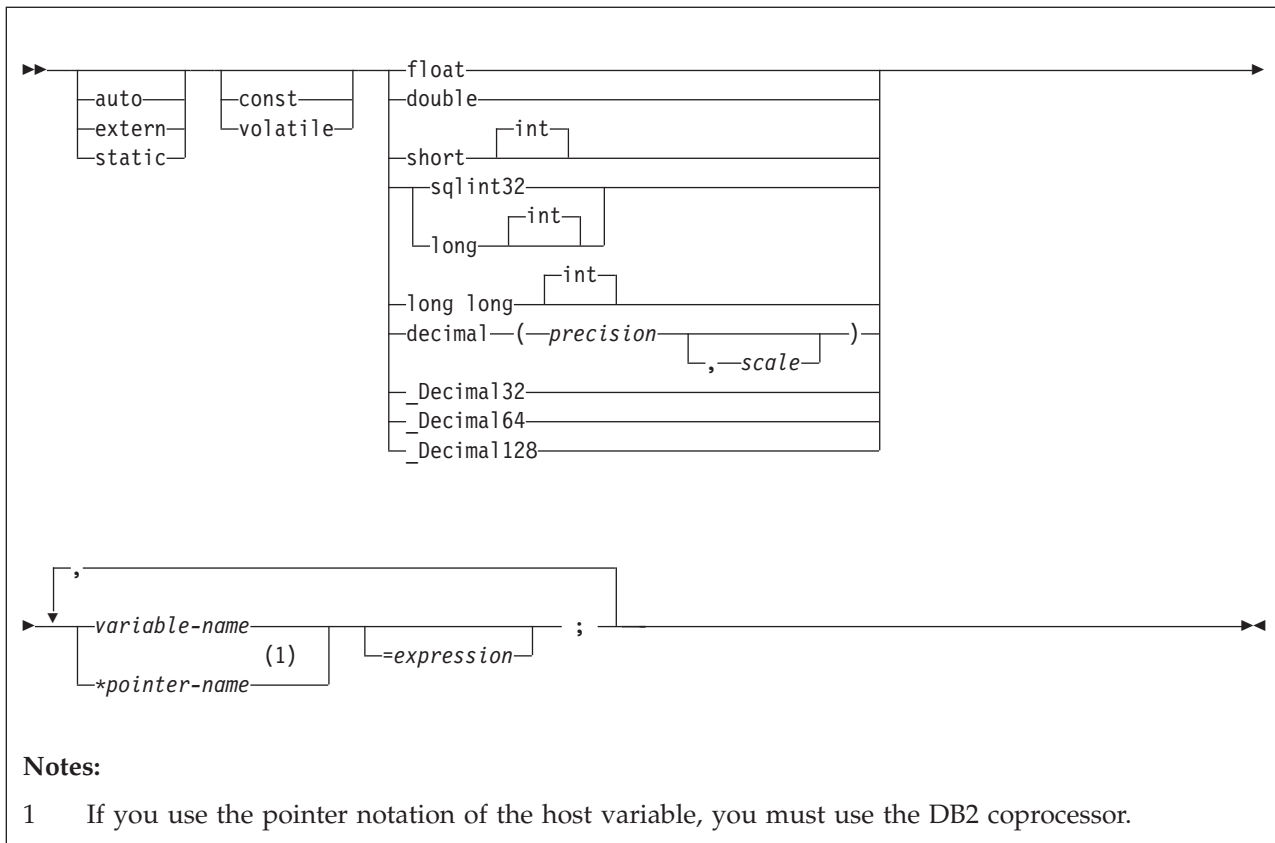
- Although DB2 allows you to use properly formed L-literals in C application programs, DB2 does not check for all the restrictions that the C compiler imposes on the L-literal. \
- Do not use L-literals in SQL statements. Use DB2 graphic string constants in SQL statements to work with the L-literal.

Recommendations:

- Be careful of overflow. For example, suppose that you retrieve an INTEGER column value into a short integer host variable, and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provide an indicator variable.
- Be careful of truncation. Ensure that the host variable that you declare can contain the data and a NUL terminator, if needed. Retrieving a floating-point or decimal column value into a long integer host variable removes any fractional part of the value.

Numeric host variables

The following diagram shows the syntax for declaring numeric host variables.



Restrictions:

- If your C compiler does not have a decimal data type, no exact equivalent exists for the SQL data type DECIMAL. In this case, you can use one of the following variables or techniques to handle decimal values:

- An integer or floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or if you want to preserve a fractional value, use floating-point variables. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.
- A character-string host variable. Use the CHAR function to get a string representation of a decimal number.
- The DECIMAL function to explicitly convert a value to a decimal data type, as shown in the following example:

```
long duration=10100; /* 1 year and 1 month */
char result_dt[11];

EXEC SQL SELECT START_DATE + DECIMAL(:duration,8,0)
          INTO :result_dt FROM TABLE1;
```

- z/OS 1.10 or above (z/OS V1R10 XL C/C++) is required to use the decimal floating-point host data type.
- The special C only 'complex floating-point' host data type is not a supported type for host variable.
- The FLOAT precompiler option does not apply to the decimal floating-point host variable types.
- To use decimal floating-point host variable, you must use the DB2 coprocessor.

For floating-point data types, use the FLOAT SQL processing option to specify whether the host variable is in IEEE binary floating-point or z/Architecture hexadecimal floating-point format. DB2 does not check if the format of the host variable contents match the format that you specified with the FLOAT SQL processing option. Therefore, you need to ensure that your floating-point host variable contents match the format that you specified with the FLOAT SQL processing option. DB2 converts all floating-point input data to z/Architecture hexadecimal floating-point format before storing it.

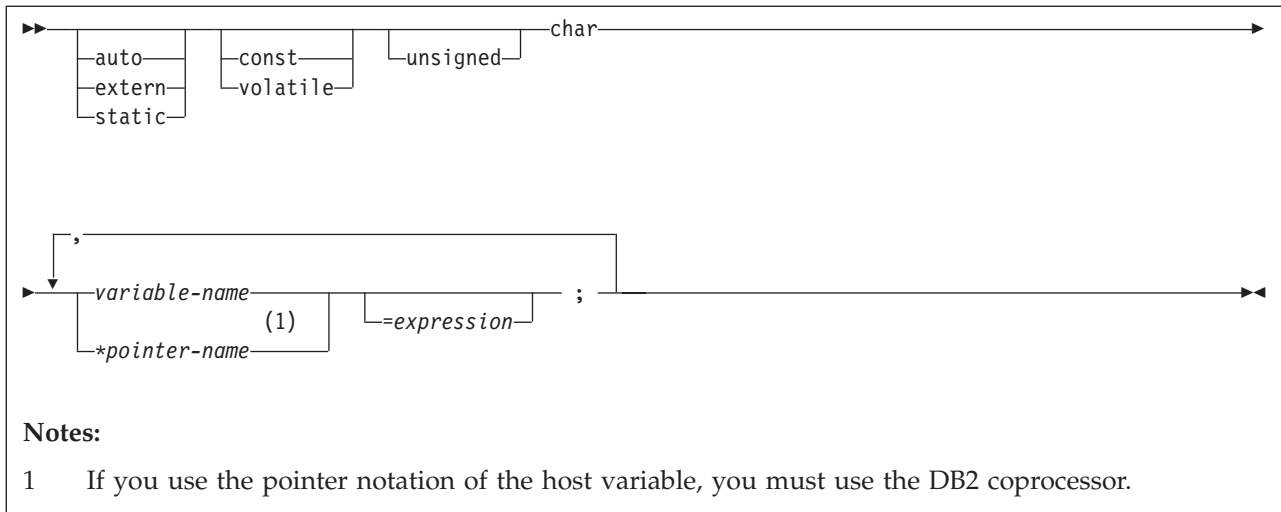
Character host variables

You can specify the following forms of character host variables:

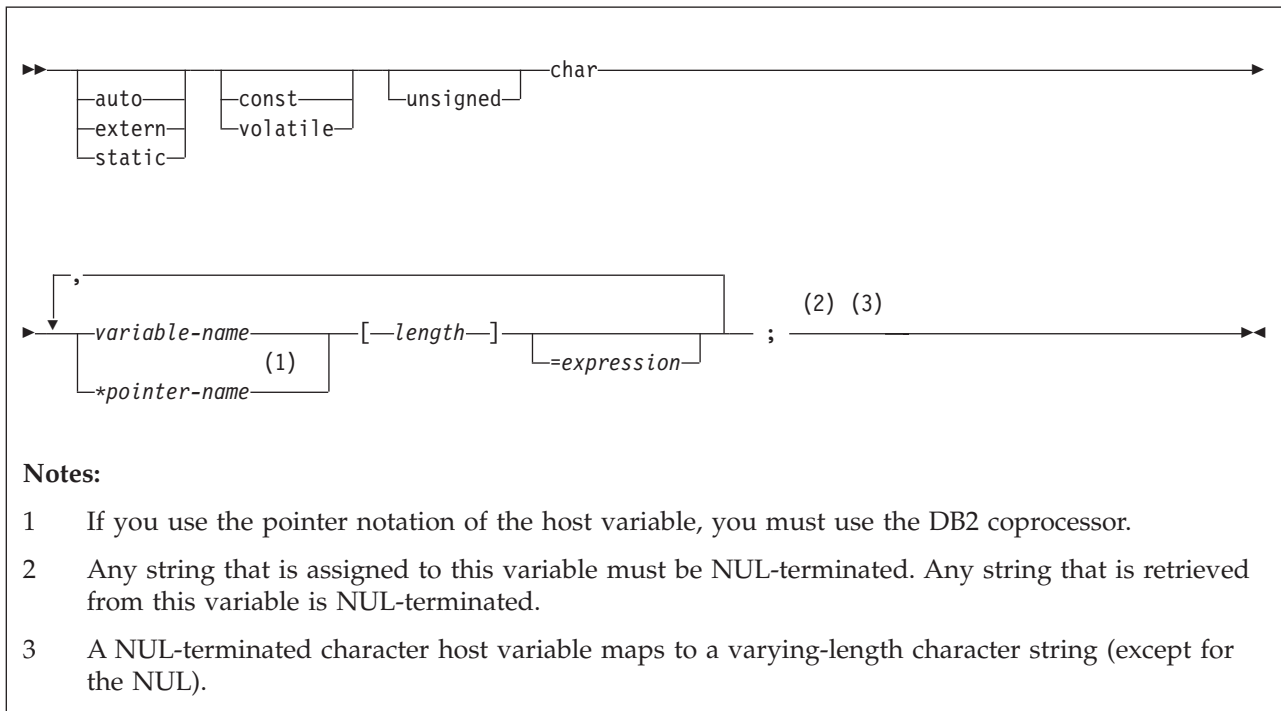
- Single-character form
- NUL-terminated character form
- VARCHAR structured form
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

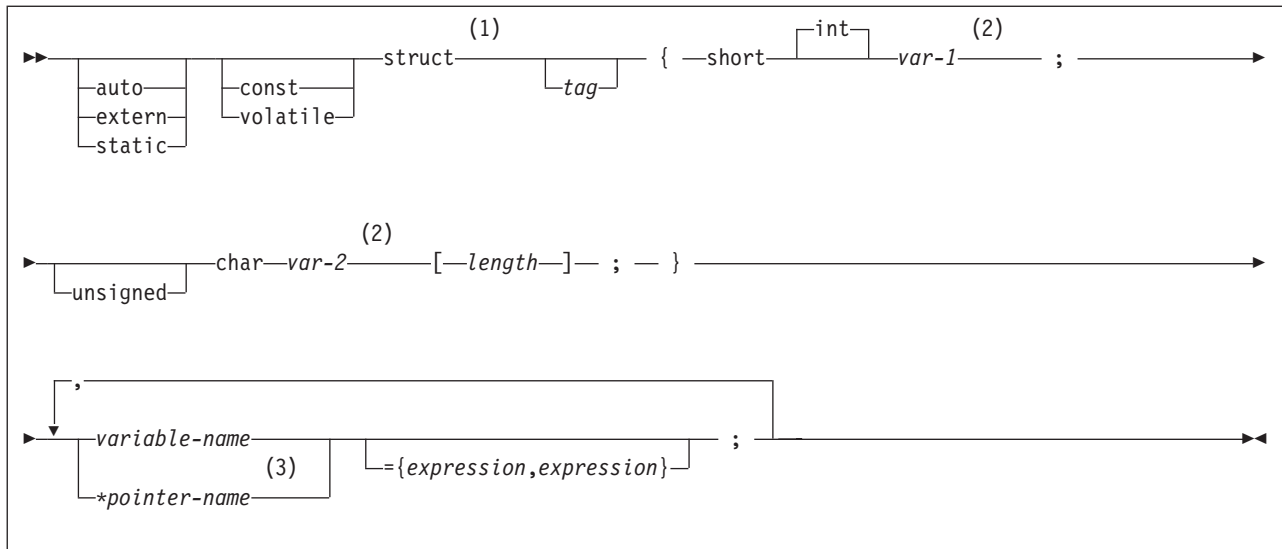
The following diagram shows the syntax for declaring single-character host variables.



The following diagram shows the syntax for declaring NUL-terminated character host variables.



The following diagram shows the syntax for declaring varying-length character host variables that use the VARCHAR structured form.



Notes:

- 1 You can use the `struct` tag to define other variables, but you cannot use them as host variables in SQL.
- 2 You cannot use `var-1` and `var-2` as host variables in an SQL statement.
- 3 If you use the pointer notation of the host variable, you must use the DB2 coprocessor.

Example: The following example code shows valid and invalid declarations of the VARCHAR structured form:

```
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable VARCHAR vstring */
struct VARCHAR {
    short len;
    char s[10];
} vstring;

/* invalid declaration of host variable VARCHAR wstring */
struct VARCHAR wstring;
```

For NUL-terminated string host variables, use the SQL processing options `PADNTSTR` and `NOPADNTSTR` to specify whether the variable should be padded with blanks. The option that you specify determines where the NUL-terminator is placed.

If you assign a string of length *n* to a NUL-terminated string host variable, the variable has one of the values that is shown in the following table.

Table 56. Value of a NUL-terminated string host variable that is assigned a string of length n

Length of the NUL-terminated string host variable	Value of the variable
Less than or equal to <i>n</i>	The source string up to a length of <i>n</i> -1 and a NUL at the end of the string. ¹
	DB2 sets <code>SQLWARN[1]</code> to <code>W</code> and any indicator variable that you provide to the original length of the source string.

Table 56. Value of a NUL-terminated string host variable that is assigned a string of length n (continued)

Length of the NUL-terminated string host variable	Value of the variable
Equal to $n+1$	The source string and a NUL at the end of the string. ¹
Greater than $n+1$ and the source is a fixed-length string	<p>If PADNTSTR is in effect The source string, blanks to pad the value, and a NUL at the end of the string.</p> <p>If NOPADNTSTR is in effect The source string and a NUL at the end of the string.</p>
Greater than $n+1$ and the source is a varying-length string	The source string and a NUL at the end of the string. ¹

Note:

1. In these cases, whether NOPADNTSTR or PADNTSTR is in effect is irrelevant.

Restriction: If you use the DB2 precompiler, you cannot use a host variable that is of the NUL-terminated form in either a PREPARE or DESCRIBE statement. However, if you use the DB2 coprocessor, you can use host variables of the NUL-terminated form in PREPARE, DESCRIBE, and EXECUTE IMMEDIATE statements.

Graphic host variables

You can specify the following forms of graphic host variables:

- Single-graphic form
- NUL-terminated graphic form
- VARGRAPHIC structured form.
- DBCLOBs

Recommendation: Instead of using the C data type `wchar_t` to define graphic and vargraphic host variables, use one of the following techniques:

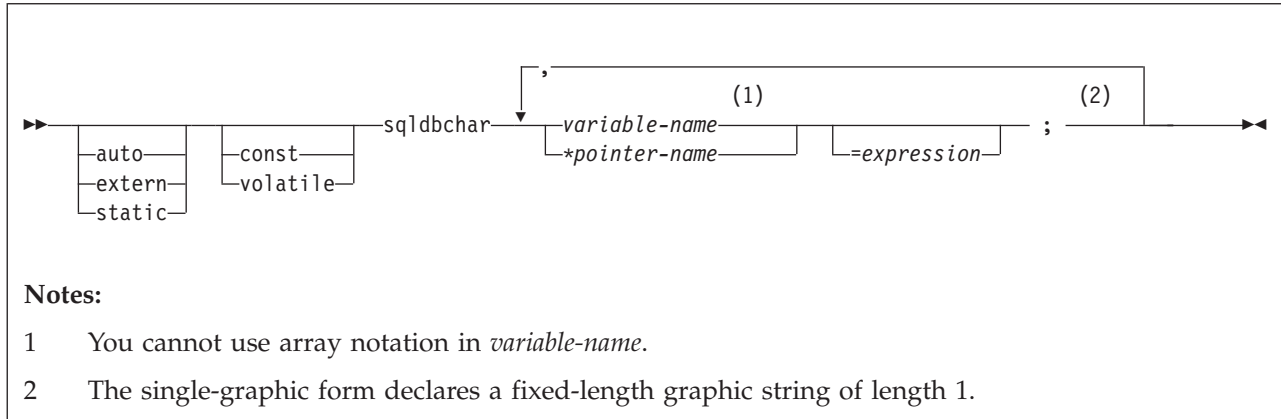
- Define the `sqldbchar` data type by using the following typedef statement:


```
typedef unsigned short sqldbchar;
```
- Use the `sqldbchar` data type that is defined in the typedef statement in one of the following files or libraries:
 - SQL library, `sql.h`
 - DB2 CLI library, `sqlcli.h`
 - SQLUDF file in data set `DSNA10.SDSNC.H`
- Use the C data type `unsigned short`.

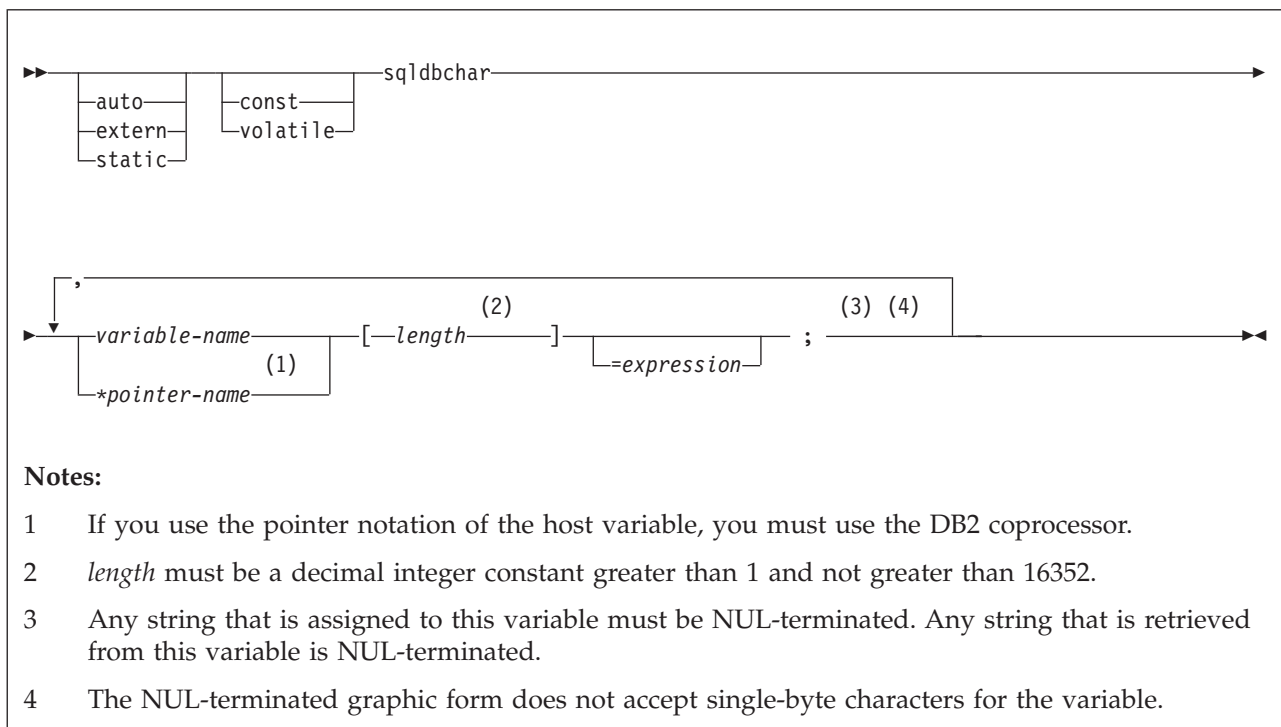
Using `sqldbchar` or `unsigned short` enables you to manipulate DBCS and Unicode UTF-16 data in the same format in which it is stored in DB2. Using `sqldbchar` also makes applications easier to port to other platforms.

The following diagrams show the syntax for forms other than DBCLOBs.

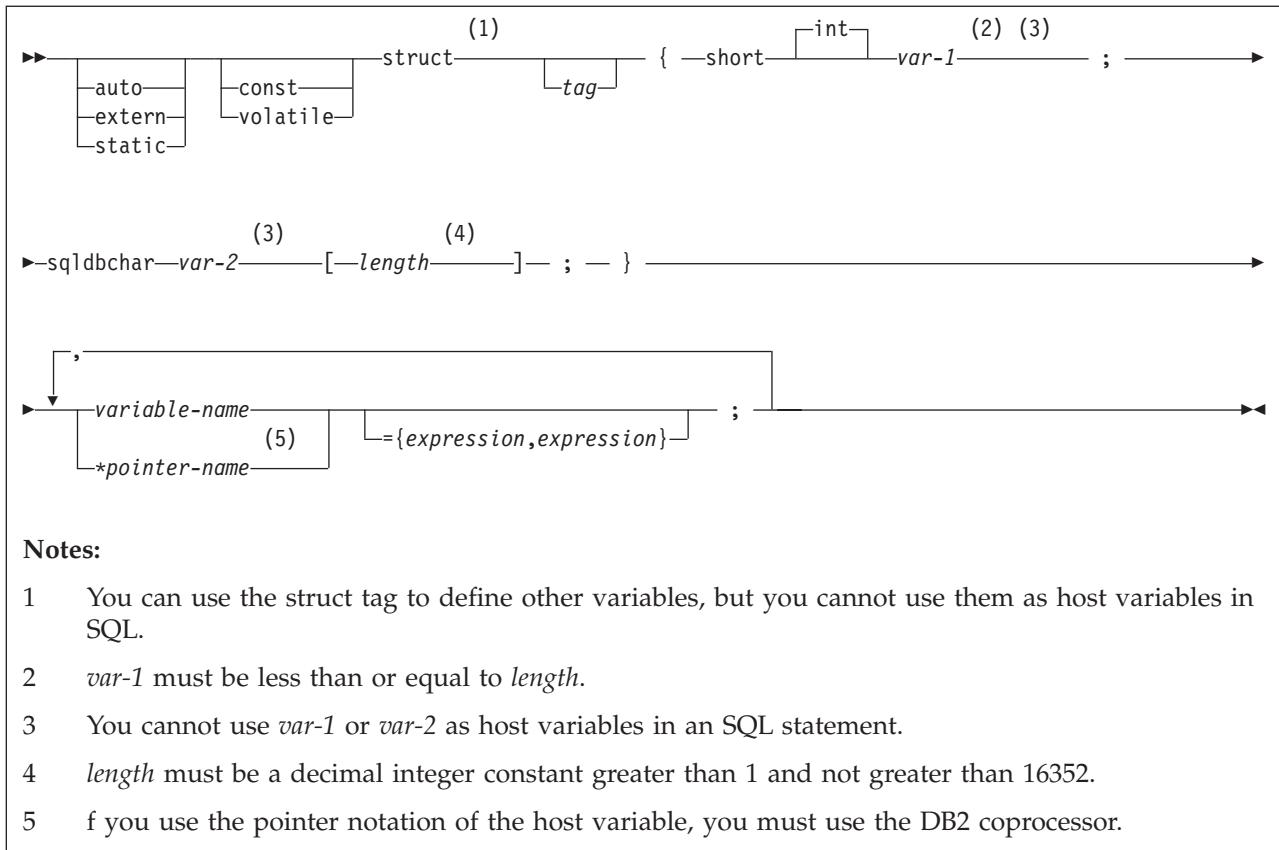
The following diagram shows the syntax for declaring single-graphic host variables.



The following diagram shows the syntax for declaring NUL-terminated graphic host variables.



The following diagram shows the syntax for declaring graphic host variables that use the VARGRAPHIC structured form.



Example: The following example shows valid and invalid declarations of graphic host variables that use the VARGRAPHIC structured form:

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
/* valid declaration of host variable structured vgraph */
struct VARGRAPH {
  short len;
  sqlbchar d[10];
} vgraph;
```

```
/* invalid declaration of host variable structured wgraph */
struct VARGRAPH wgraph;
```

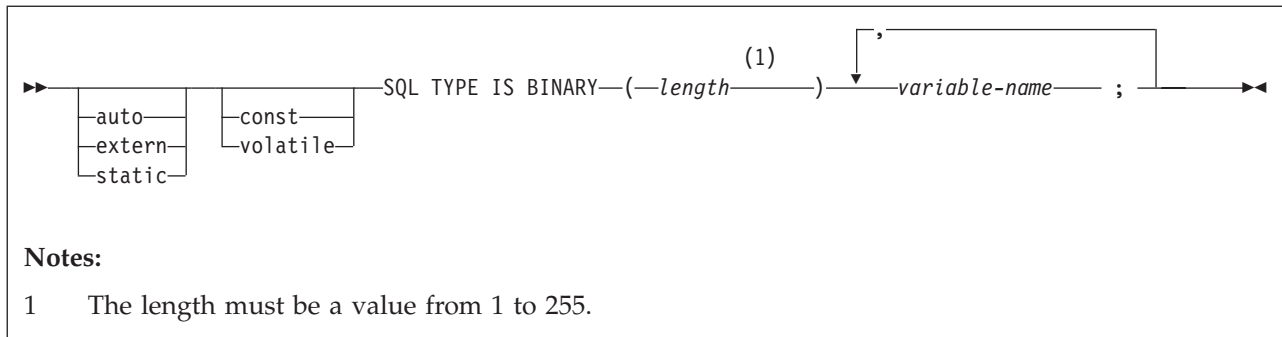
Binary host variables

You can specify the following forms of binary host variables:

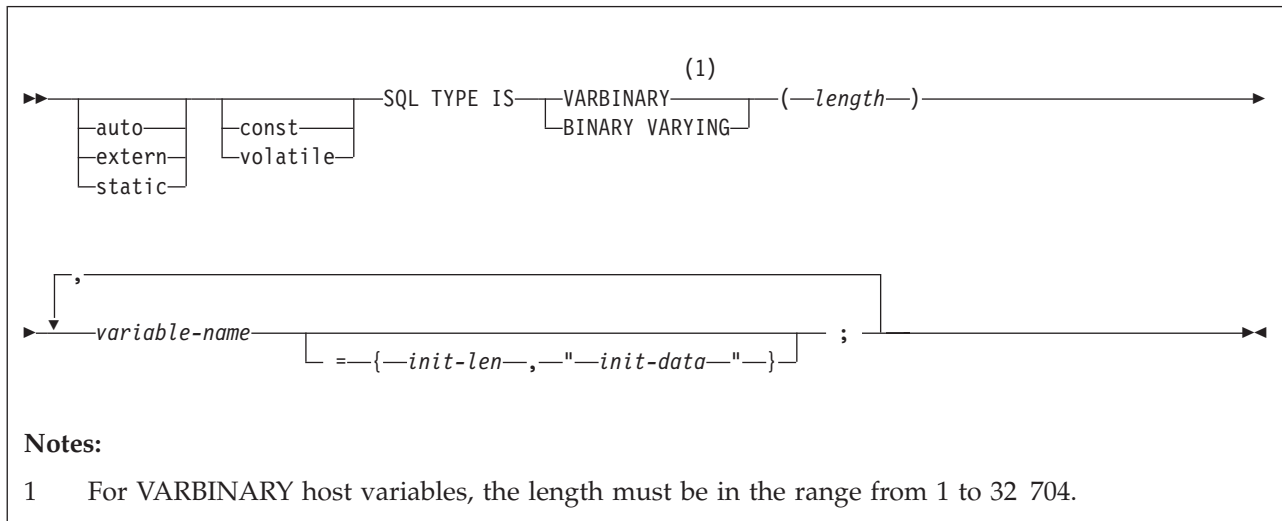
- Fixed-length strings
- Varying-length strings
- BLOBs

The following diagrams show the syntax for forms other than BLOBs.

The following diagram shows the syntax for declaring binary host variables.



The following diagram shows the syntax for declaring VARBINARY host variables.



The C language does not have variables that correspond to the SQL binary data types BINARY and VARBINARY. 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 the C language structure in the output source member.

When you reference a BINARY or VARBINARY host variable in an SQL statement, you must use the variable that you specify in the SQL TYPE declaration. When you reference the host variable in a host language statement, you must use the variable that DB2 generates.

Examples of binary variable declarations: The following table shows examples of variables that DB2 generates when you declare binary host variables.

Table 57. Examples of BINARY and VARBINARY variable declarations for C

Variable declaration that you include in your C program	Corresponding variable that DB2 generates in the output source member
SQL TYPE IS BINARY(10) bin_var;	char bin_var[10]
SQL TYPE IS VARBINARY(10) vbin_var;	struct { short length; char data[10]; } vbin_var;

Recommendation: Be careful when you use binary host variables with C and C++. The SQL TYPE declaration for BINARY and VARBINARY does not account for the NUL-terminator that C expects, because binary strings are not NUL-terminated strings. Also, the binary host variable might contain zeroes at any point in the string.

Result set locators

The following diagram shows the syntax for declaring result set locators.

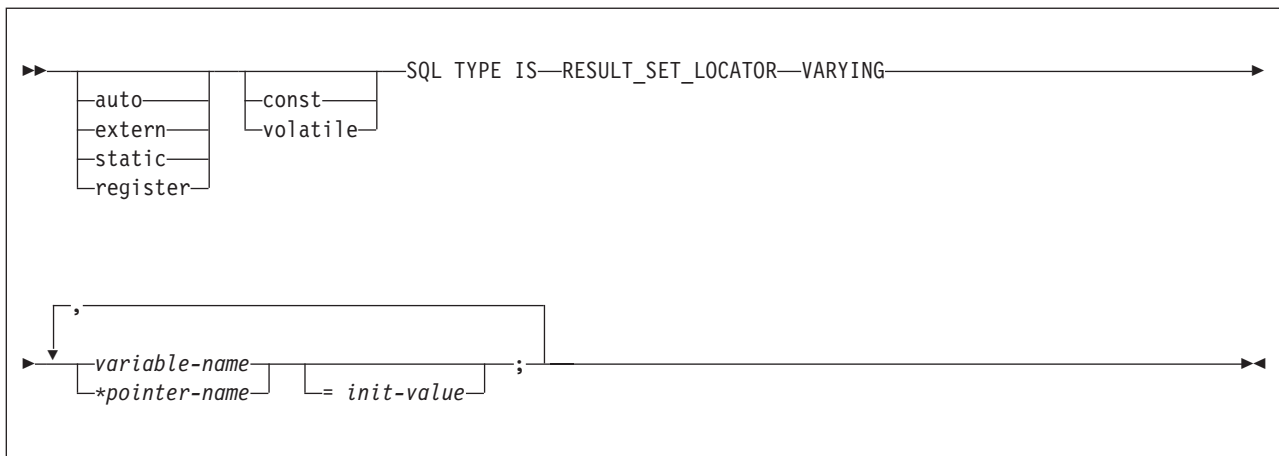
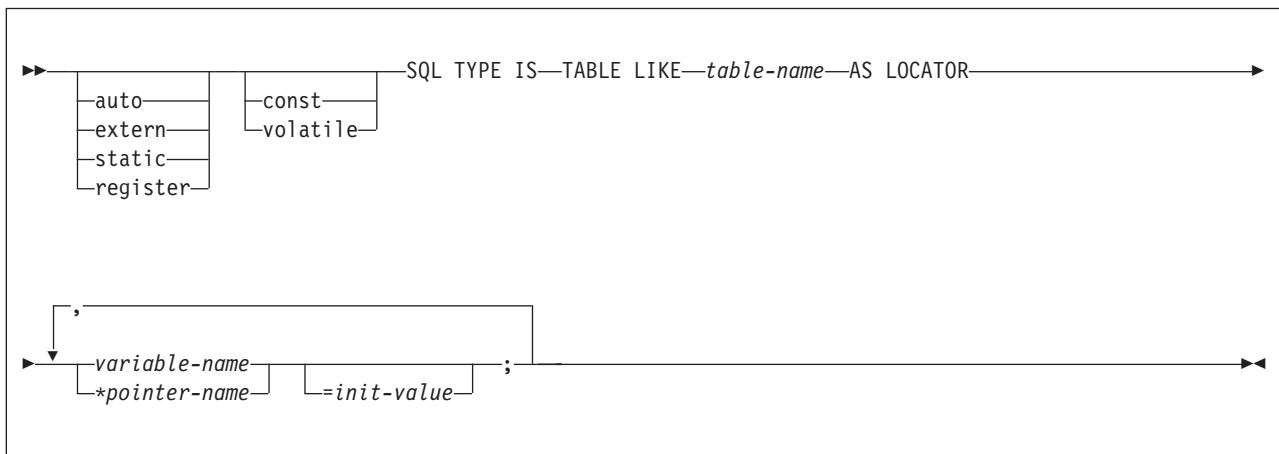


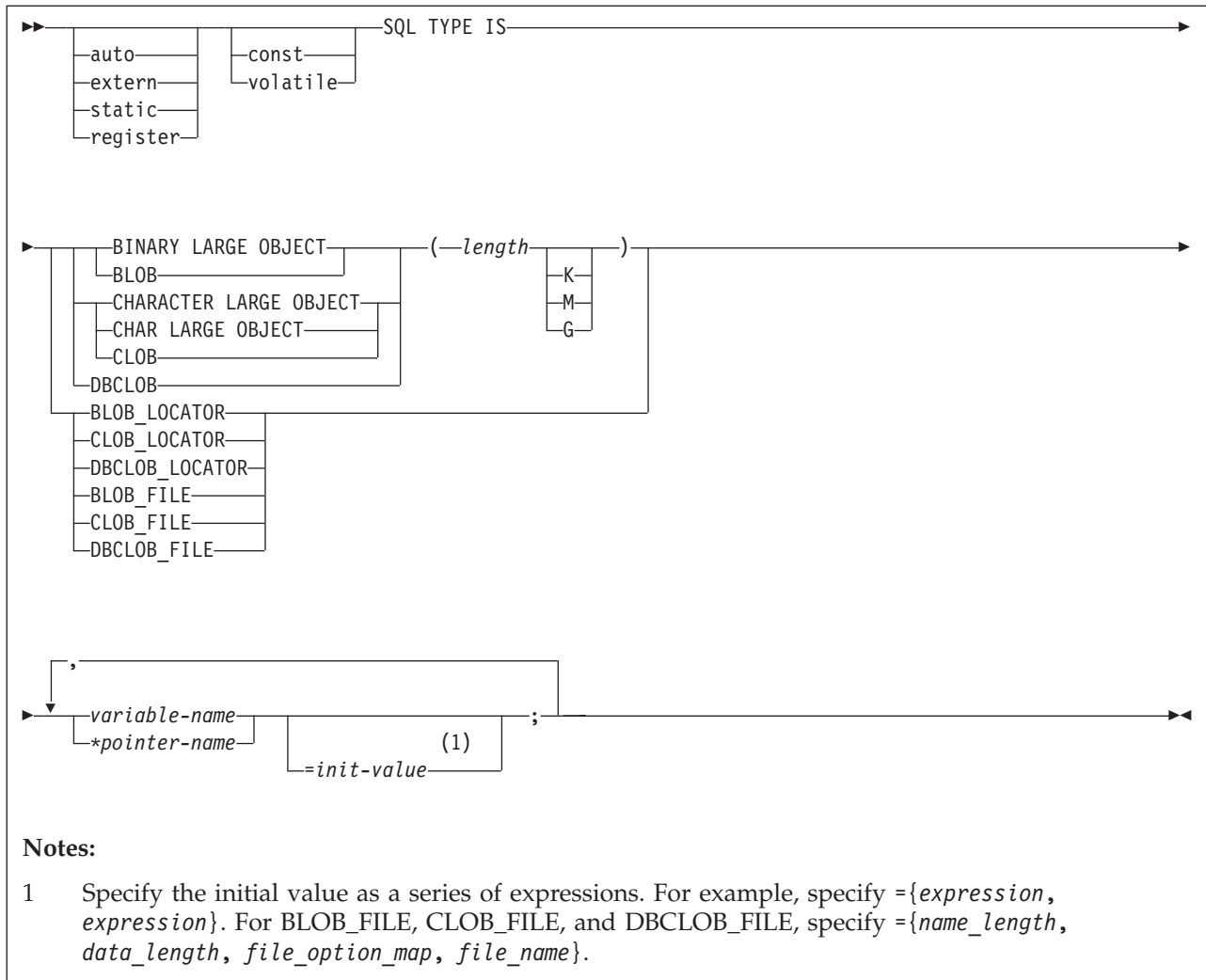
Table locators

The following diagram shows the syntax for declaring table locators.



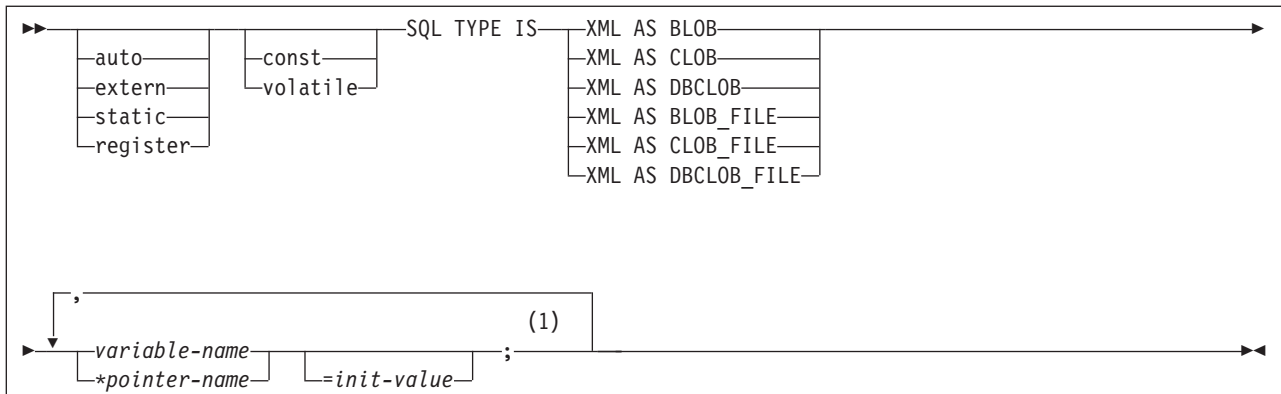
LOB variables, locators, and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables, locators, and file reference variables.



XML data host and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.

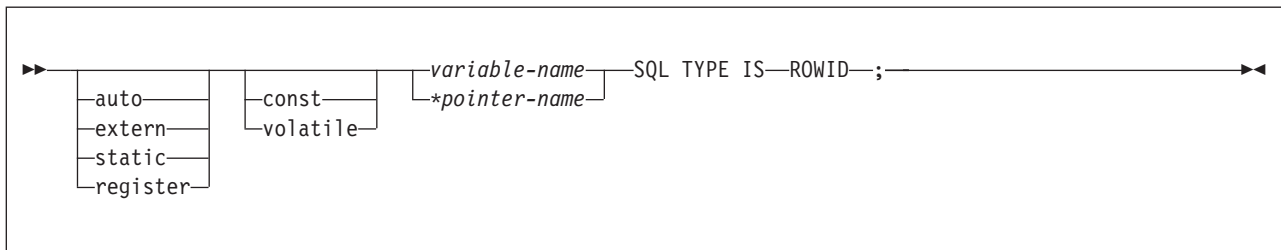


Notes:

- 1 Specify the initial value as a series of expressions. For example, specify `={expression, expression}`. For BLOB_FILE, CLOB_FILE, and DBCLOB_FILE, specify `={name_length, data_length, file_option_map, file_name}`.

ROWID host variables

The following diagram shows the syntax for declaring ROWID host variables.



Constants

The syntax for constants in C and C++ programs differs from the syntax for constants in SQL statements in the following ways:

- C/C++ uses various forms for numeric literals (possible suffixes are: ll, LL, u, U, f, F, l, L, df, DF, dd, DD, dl, DL, d, D). For example, in C/C++:
 - 4850976 is a decimal literal
 - 0x4bD is a hexadecimal integer literal
 - 03245 is an octal integer literal
 - 3.2E+4 is a double floating-point literal
 - 3.2E+4f is a float floating-point literal
 - 3.2E+4l is a long double floating-point literal
 - 0x4bDP+4 is a double hexadecimal floating-point literal
 - 22.2df is a `_Decimal32` decimal floating-point literal
 - 0.00D is a fixed-point decimal literal (z/OS only when `LANGLVL(EXTENDED)` is specified)
- Use C/C++ literal form only outside of SQL statements. Within SQL statements, use numeric constants.

- In C, character constants and string constants can use escape sequences. You cannot use the escape sequences in SQL statements.
- Apostrophes and quotation marks have different meanings in C and SQL. In C, you can use double quotation marks to delimit string constants, and apostrophes to delimit character constants.

Example of the use of quotation marks in C:

```
printf( "%d lines read. \n", num_lines);
```

Example of the use of apostrophes in C:

```
#define NUL '\0'
```

In SQL, you can use double quotation marks to delimit identifiers and apostrophes to delimit string constants.

Example of the use of quotation marks in SQL:

```
SELECT "COL#1" FROM TBL1;
```

Example of the use of apostrophes in SQL:

```
SELECT COL1 FROM TBL1 WHERE COL2 = 'BELL';
```

- Character data in SQL is distinct from integer data. Character data in C is a subtype of integer data.

Related concepts:

“Host variables” on page 174

“Rules for host variables in an SQL statement” on page 183

“Large objects (LOBs)” on page 465

Related tasks:

“Determining whether a retrieved value in a host variable is null or truncated” on page 186

“Inserting a single row by using a host variable” on page 189

“Inserting null values into columns by using indicator variables or arrays” on page 190

“Retrieving a single row of data into host variables” on page 184

“Retrieving a single row of data into a host structure” on page 193

“Updating data by using host variables” on page 189

Related reference:

“Descriptions of SQL processing options” on page 932

Host variable arrays in C

In C and C++ programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Restrictions:

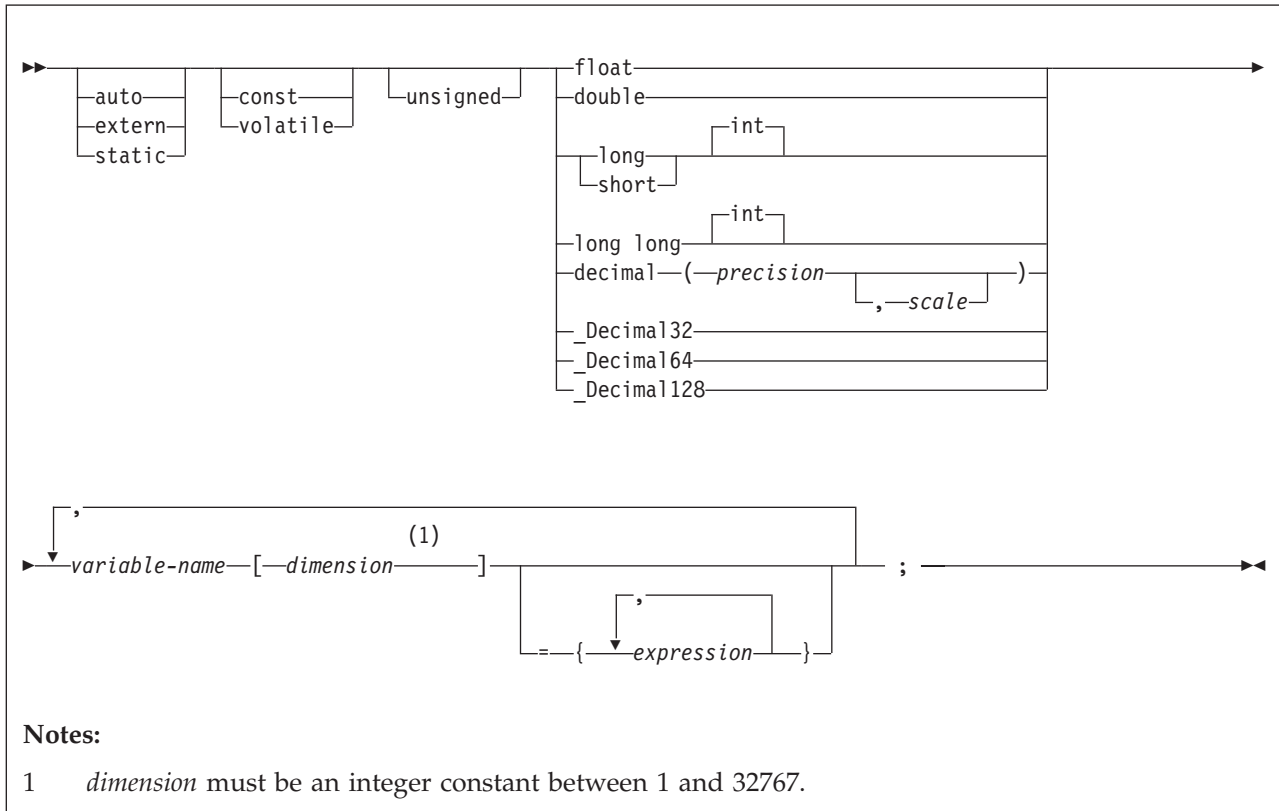
- Only some of the valid C declarations are valid host variable array declarations. If the declaration for a variable array is not valid, any SQL statement that references the variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.
- For both C and C++, you cannot specify the `_packed` attribute on the structure declarations for the following arrays that are used in multiple-row INSERT, FETCH, and MERGE statements:

- varying-length character arrays
- varying-length graphic arrays
- LOB arrays

In addition, the #pragma pack(1) directive cannot be in effect if you plan to use these arrays in multiple-row statements.

Numeric host variable arrays

The following diagram shows the syntax for declaring numeric host variable arrays.



Example: The following example shows a declaration of a numeric host variable array:

```
EXEC SQL BEGIN DECLARE SECTION;
    /* declaration of numeric host variable array */
    long serial_num[10];
    ...
EXEC SQL END DECLARE SECTION;
```

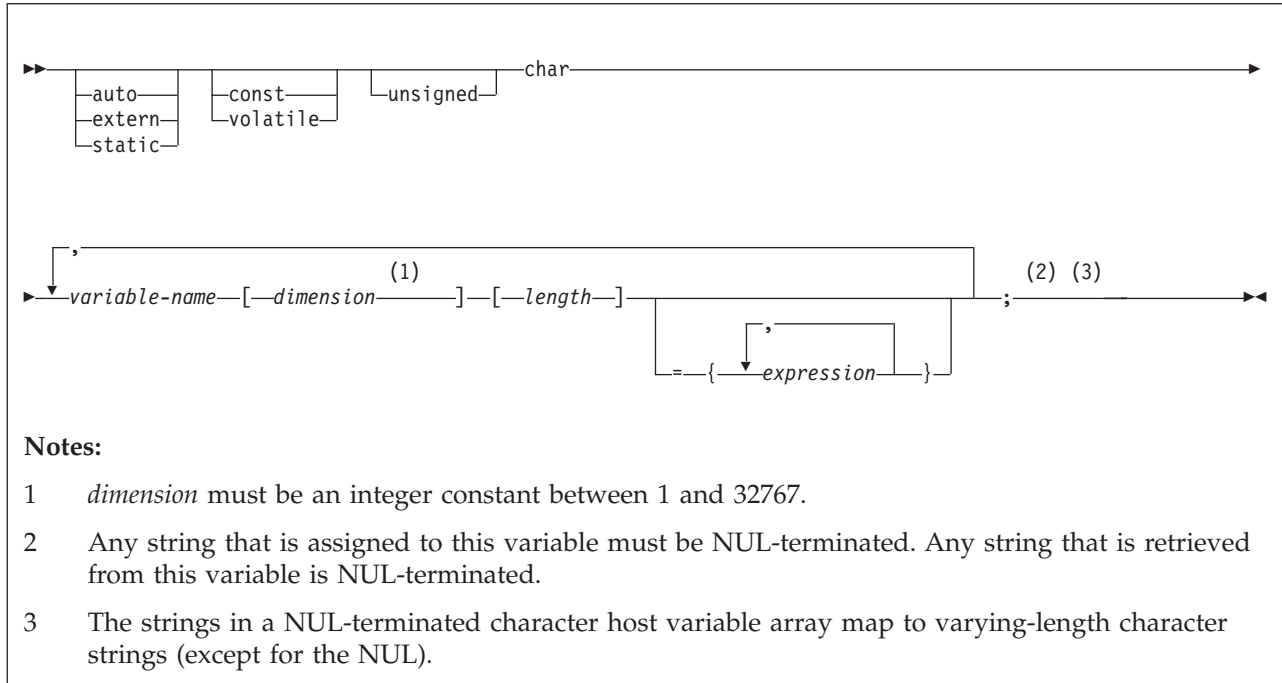
Character host variable arrays

You can specify the following forms of character host variable arrays:

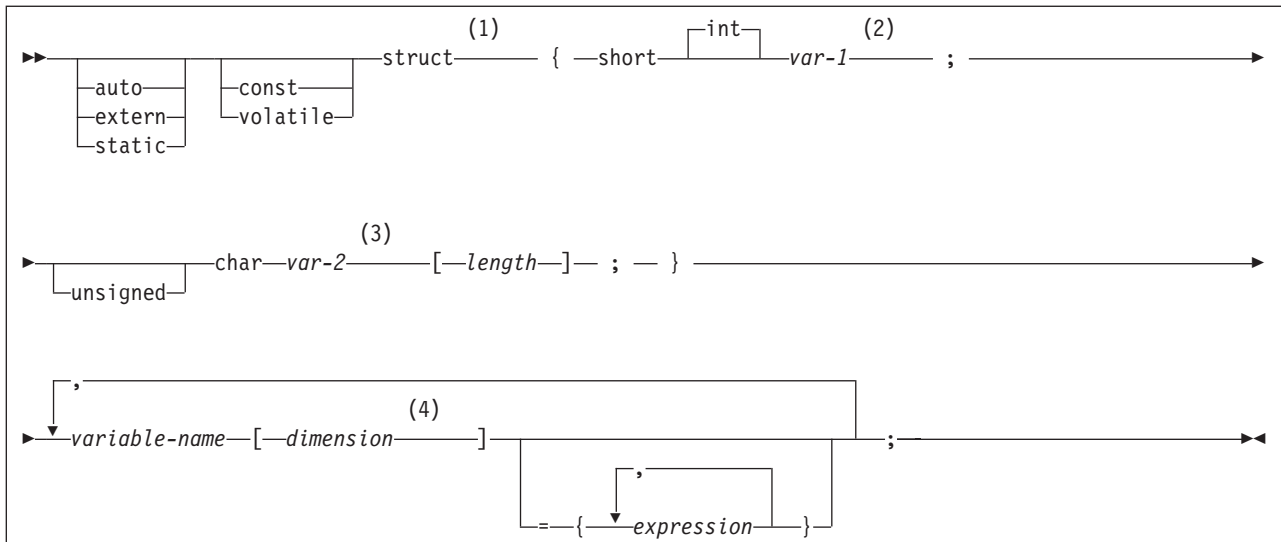
- NUL-terminated character form
- VARCHAR structured form
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring NUL-terminated character host variable arrays.



The following diagram shows the syntax for declaring varying-length character host variable arrays that use the VARCHAR structured form.



Notes:

- 1 You can use the struct tag to define other variables, but you cannot use them as host variable arrays in SQL.
- 2 *var-1* must be a scalar numeric variable.
- 3 *var-2* must be a scalar CHAR array variable.
- 4 *dimension* must be an integer constant between 1 and 32767.

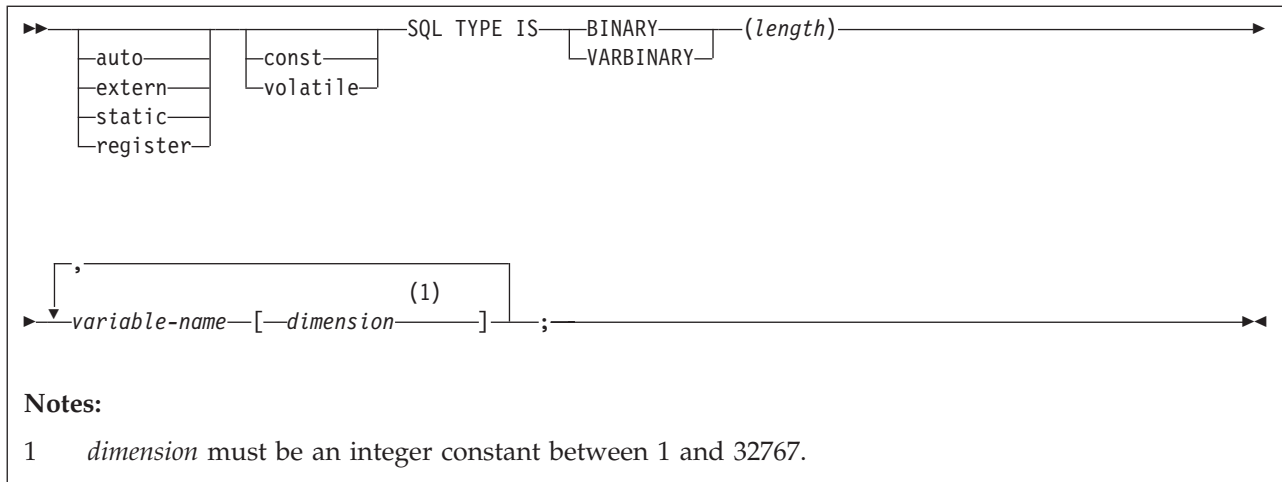
Example: The following example shows valid and invalid declarations of VARCHAR host variable arrays.

```
EXEC SQL BEGIN DECLARE SECTION;
/* valid declaration of VARCHAR host variable array */
struct VARCHAR {
  short len;
  char s[18];
} name[10];

/* invalid declaration of VARCHAR host variable array */
struct VARCHAR name[10];
```

Binary host variable arrays

The following diagram shows the syntax for declaring binary host variable arrays.



Graphic host variable arrays

You can specify the following forms of graphic host variable arrays:

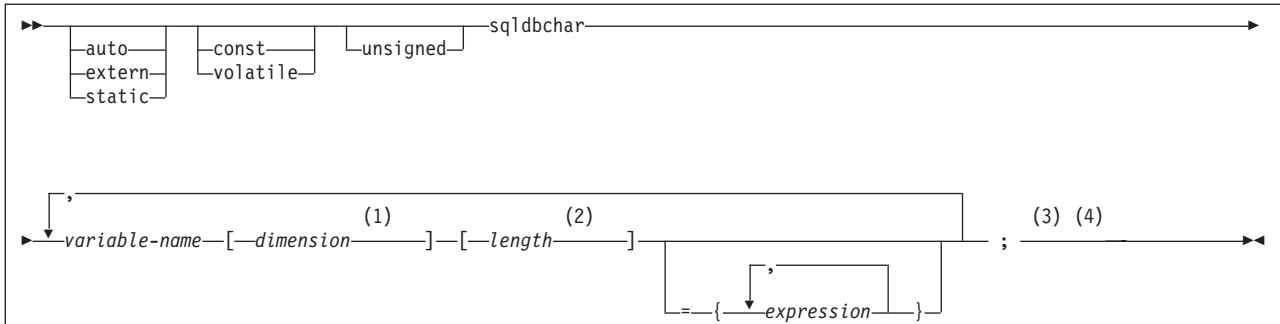
- NUL-terminated graphic form
- VARGRAPHIC structured form.

Recommendation: Instead of using the C data type `wchar_t` to define graphic and vargraphic host variable arrays, use one of the following techniques:

- Define the `sqldbcchar` data type by using the following typedef statement:

```
typedef unsigned short sqldbcchar;
```
- Use the `sqldbcchar` data type that is defined in the typedef statement in the header files that are supplied by DB2.
- Use the C data type `unsigned short`.

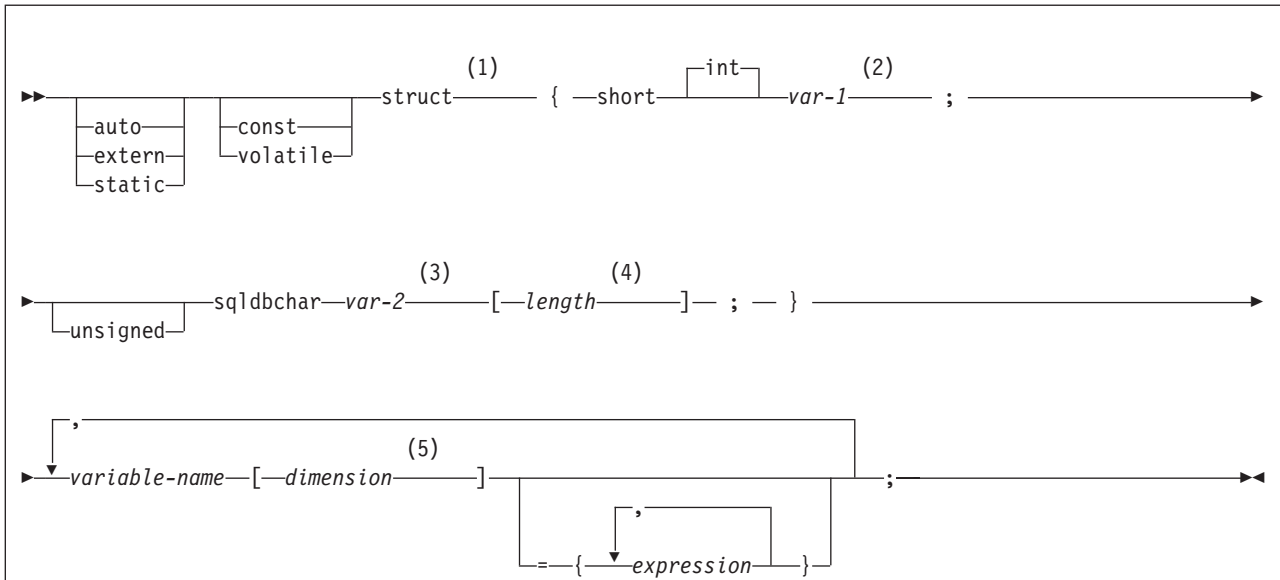
The following diagram shows the syntax for declaring NUL-terminated graphic host variable arrays.



Notes:

- 1 *dimension* must be an integer constant between 1 and 32767.
- 2 *length* must be a decimal integer constant greater than 1 and not greater than 16352.
- 3 Any string that is assigned to this variable must be NUL-terminated. Any string that is retrieved from this variable is NUL-terminated.
- 4 Do not assign single-byte characters into a NUL-terminated graphic host variable array

The following diagram shows the syntax for declaring graphic host variable arrays that use the VARGRAPHIC structured form.



Notes:

- 1 You can use the struct tag to define other variables, but you cannot use them as host variable arrays in SQL.
- 2 *var-1* must be a scalar numeric variable.
- 3 *var-2* must be a scalar char array variable.
- 4 *length* must be a decimal integer constant greater than 1 and not greater than 16352.
- 5 *dimension* must be an integer constant between 1 and 32767.

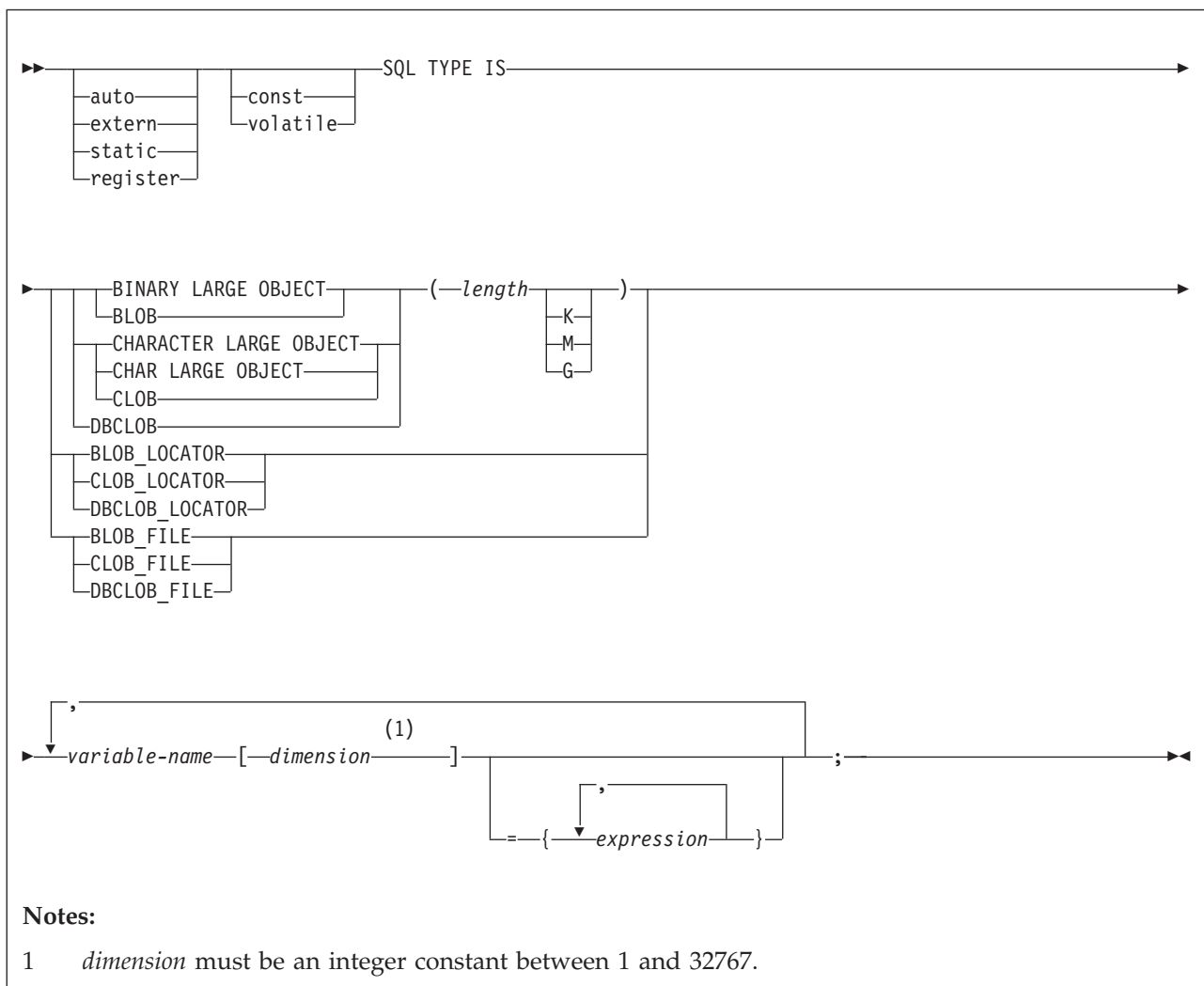
Example: The following example shows valid and invalid declarations of graphic host variable arrays that use the VARGRAPHIC structured form.

```
EXEC SQL BEGIN DECLARE SECTION;
/* valid declaration of host variable array vgraph */
struct VARGRAPH {
    short len;
    sqldbcchar d[10];
} vgraph[20];

/* invalid declaration of host variable array vgraph */
struct VARGRAPH vgraph[20];
```

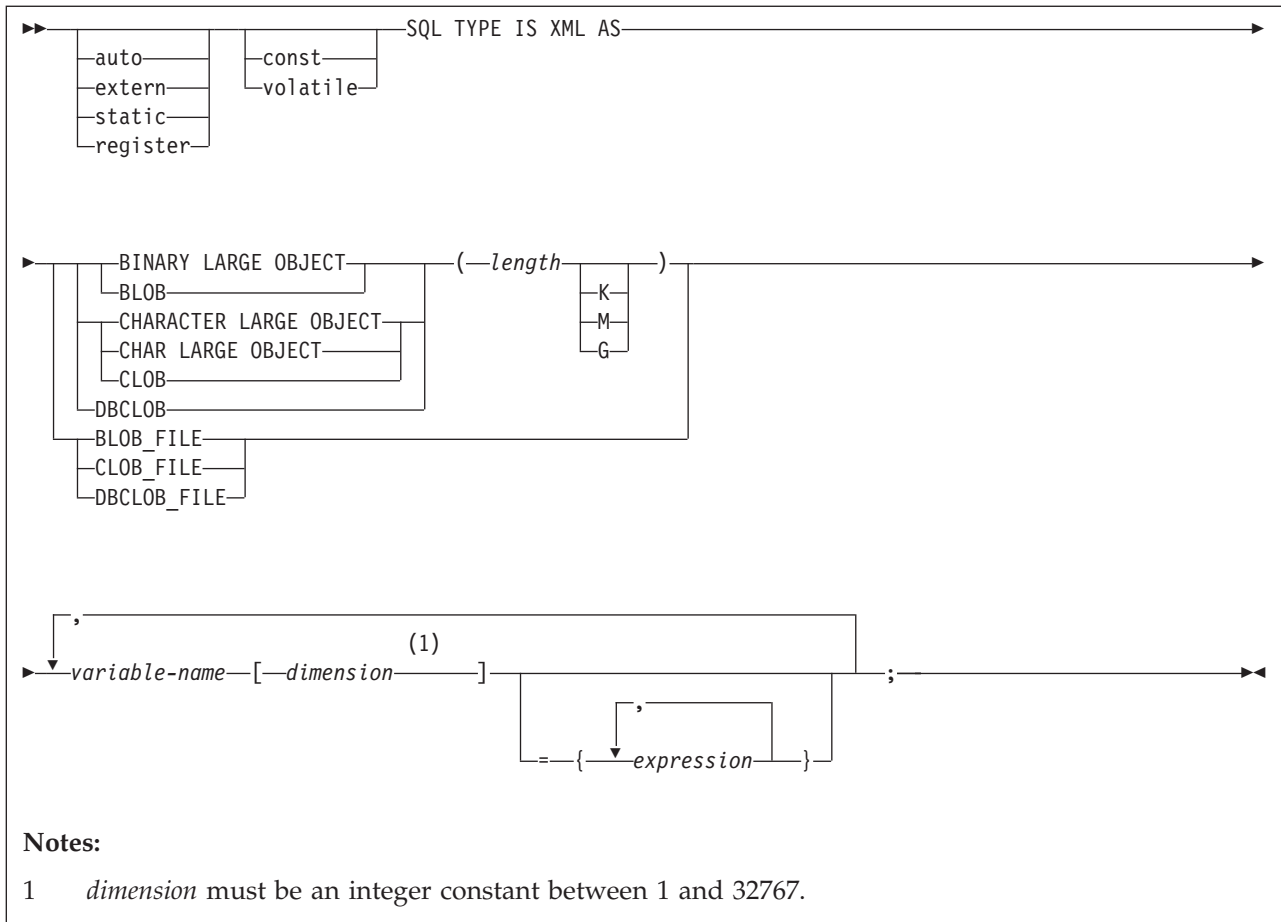
LOB, locator, and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable arrays, locators, and file reference variables.



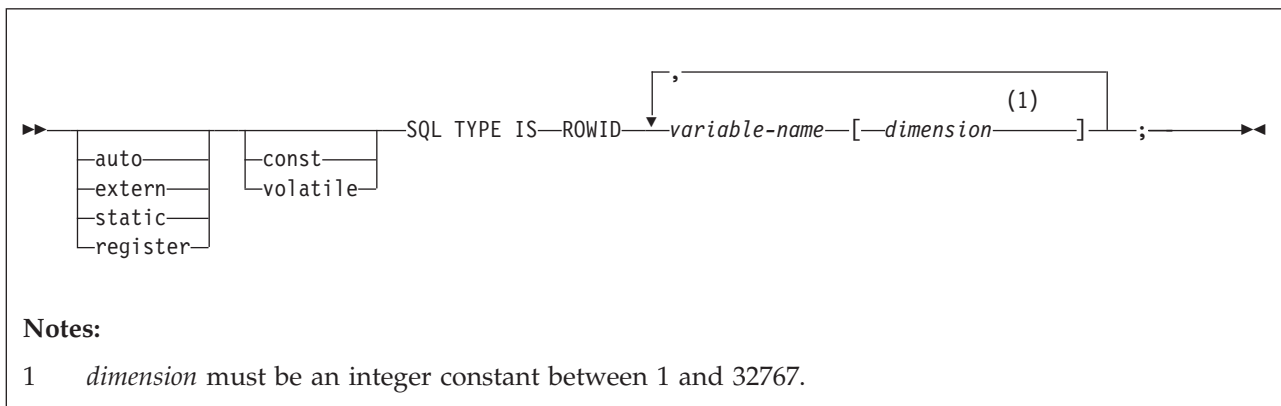
XML host and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable arrays and file reference variable arrays for XML data types.



ROWID variable arrays

The following diagram shows the syntax for declaring ROWID variable arrays.



Related concepts:

“Host variable arrays in an SQL statement” on page 191

“Host variable arrays” on page 175

“Large objects (LOBs)” on page 465

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

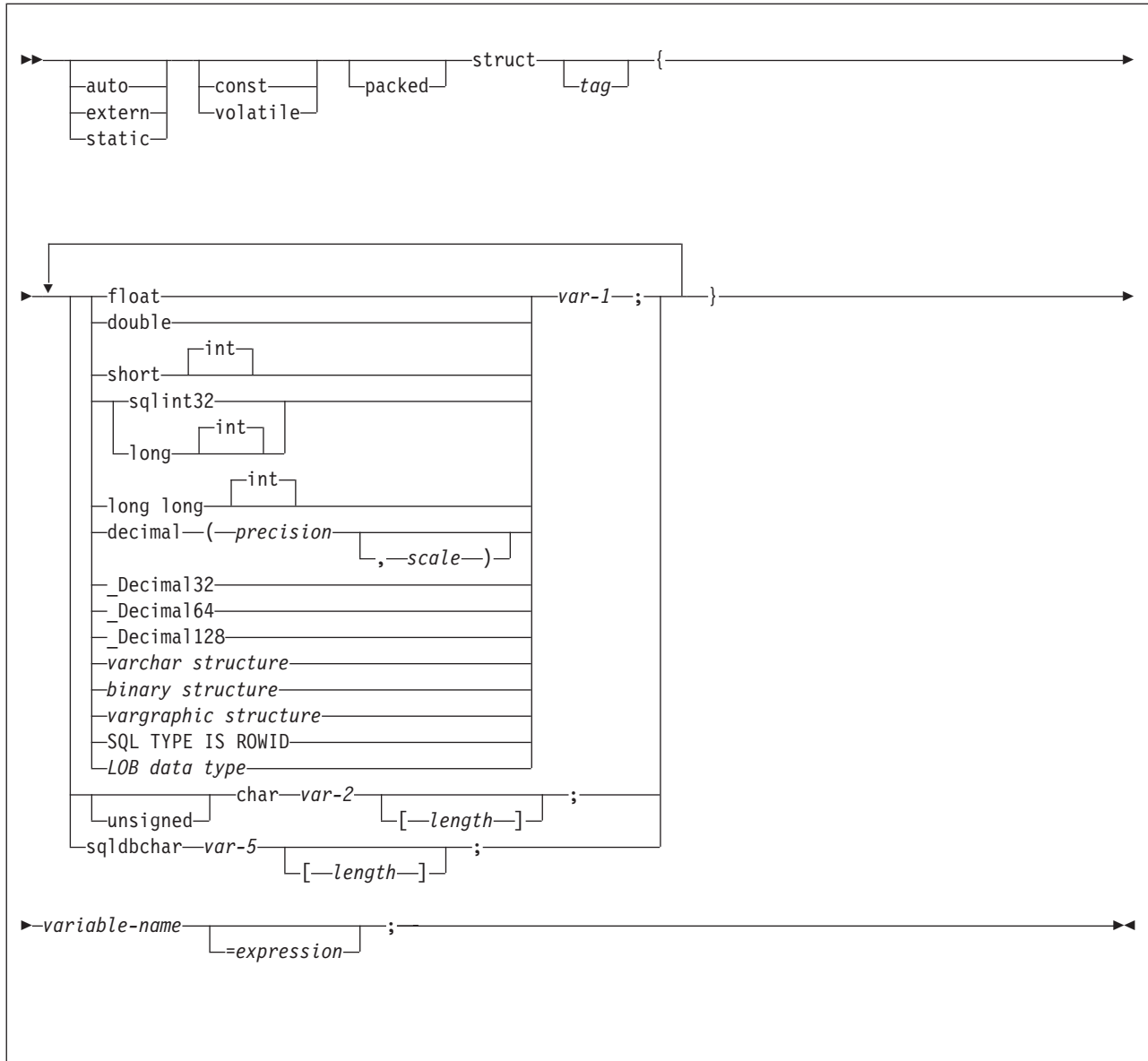
“Retrieving multiple rows of data into host variable arrays” on page 192

Host structures in C

A C host structure contains an ordered group of data fields.

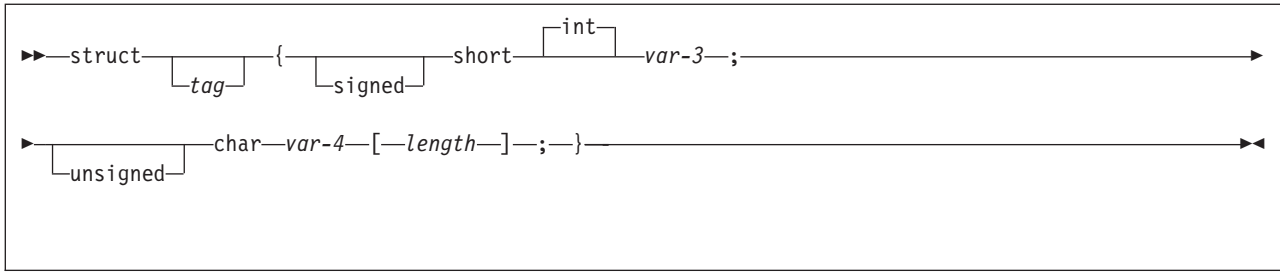
Host structures

The following diagram shows the syntax for declaring host structures.



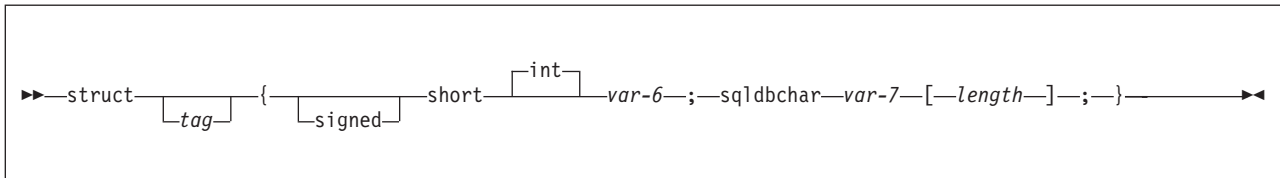
VARCHAR structures

The following diagram shows the syntax for VARCHAR structures that are used within declarations of host structures.



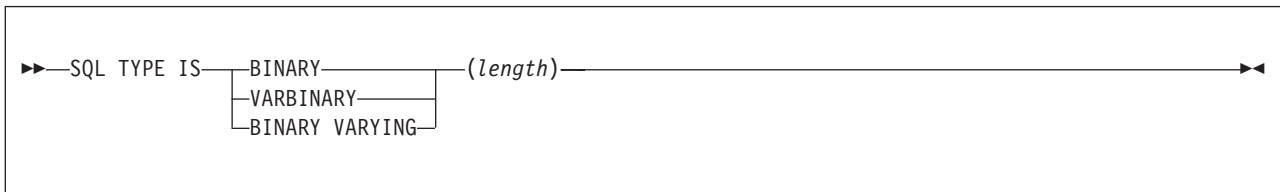
VARGRAPHIC structures

The following diagram shows the syntax for VARGRAPHIC structures that are used within declarations of host structures.



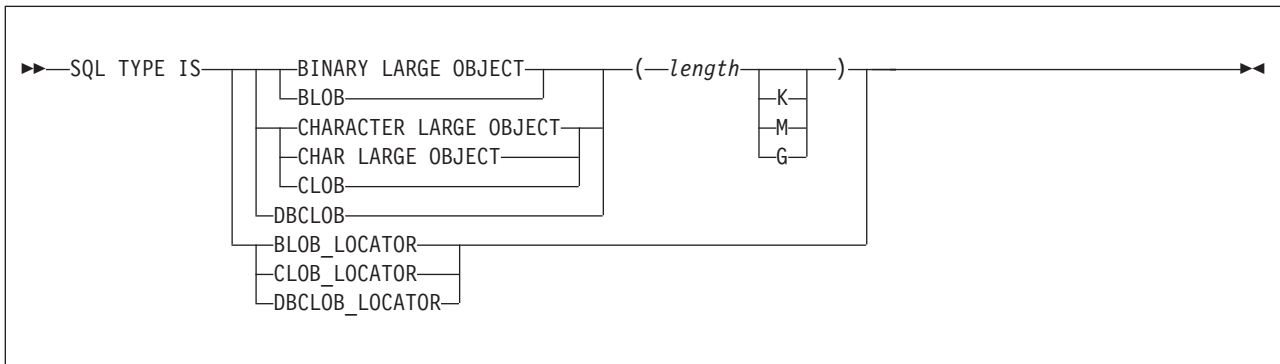
Binary structures

The following diagram shows the syntax for binary structures that are used within declarations of host structures.



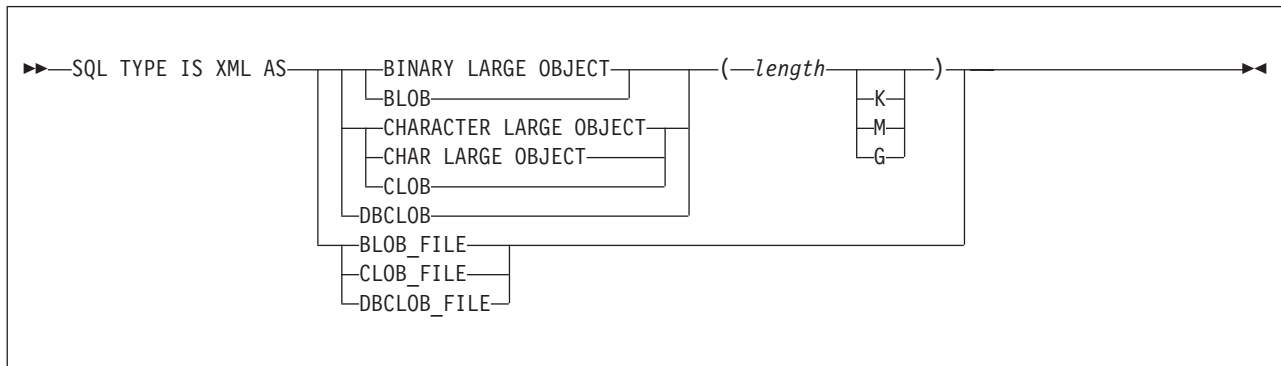
LOB data types

The following diagram shows the syntax for LOB data types that are used within declarations of host structures.



LOB data types for XML data

The following diagram shows the syntax for LOB data types that are used within declarations of host structures for XML data.



Example

In the following example, the host structure is named `target`, and it contains the fields `c1`, `c2`, and `c3`. `c1` and `c3` are character arrays, and `c2` is a host variable that is equivalent to the SQL `VARCHAR` data type. The `target` host structure can be part of another host structure but must be the deepest level of the nested structure.

```
struct {char c1[3];
        struct {short len;
                char data[5];
                }c2;
        char c3[2];
    }target;
```

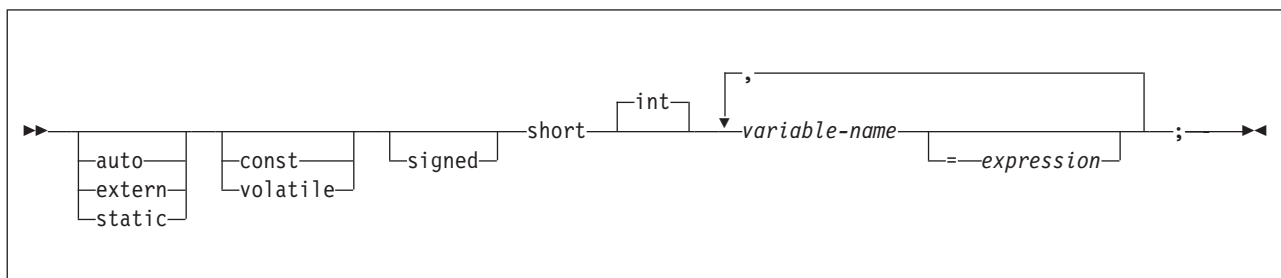
Related concepts:

"Host structures" on page 175

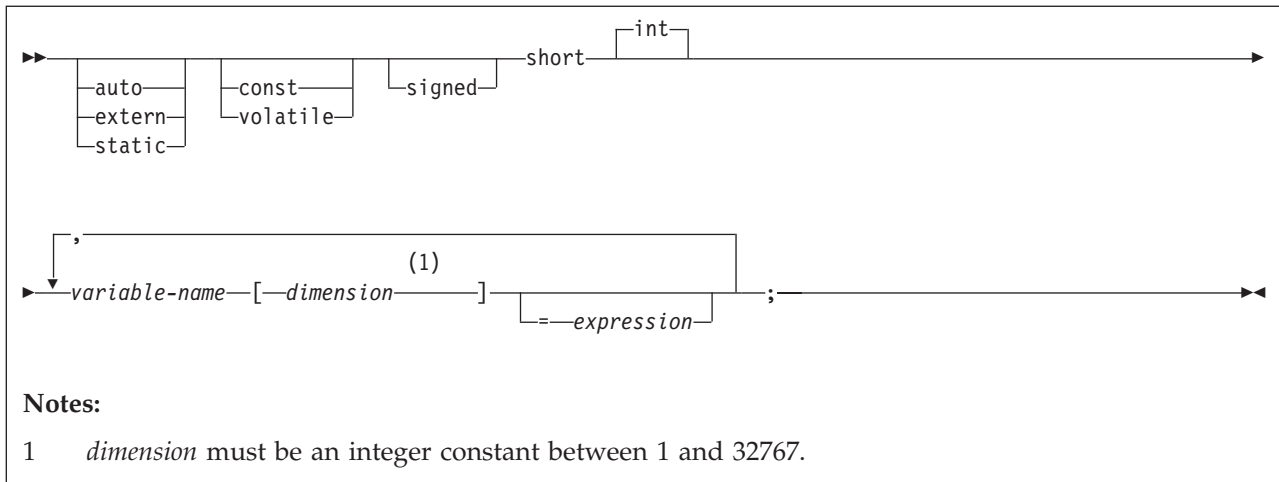
Indicator variables, indicator arrays, and host structure indicator arrays in C

An indicator variable is a 2-byte integer (short int). An indicator variable array is an array of 2-byte integers (short int). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in C and C++.



The following diagram shows the syntax for declaring an indicator array or a host structure indicator array in C and C++.



Example

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

```
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
                                :Day :DayInd,
                                :Bgn :BgnInd,
                                :End :EndInd;
```

You can declare these variables 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;
```

Related concepts:

"Indicator variables, arrays, and structures" on page 176

Related tasks:

"Inserting null values into columns by using indicator variables or arrays" on page 190

Referencing pointer host variables in C programs

If you use the DB2 coprocessor, you can reference any declared pointer host variables in your SQL statements.

To reference pointer host variables in C and C++ programs:

Specify the pointer host variable exactly as it was declared. The only exception is when you reference pointers to nul-terminated character arrays. In this case, you do not have to include the parentheses that were part of the declaration.

Examples of scalar pointer host variable references:

Table 58. Example references to scalar pointer host variables

Declaration	Description	Reference
short *hvshortp;	hvshortp is a pointer host variable that points to two bytes of storage.	EXEC SQL set:*hvshortp=123;
double *hvdoubp;	hvdoubp is a pointer host variable that points to eight bytes of storage.	EXEC SQL set:*hvdoubp=456;
char (*hvcharpn) [20];	hvcharpn is a pointer host variable that points to a nul-terminated character array of up to 20 bytes.	EXEC SQL set: *hvcharpn='nul_terminated';

Example of a bounded character pointer host variable reference: Suppose that your program declares the following bounded character pointer host variable:

```
struct {
    unsigned long len;
    char * data;
} hvbcharp;
```

The following example references this bounded character pointer host variable:

```
hvcharp.len = dynlen; a
hvcharp.data = (char *) malloc (hvcharp.len); b
EXEC SQL set :hvcharp = 'data buffer with length'; c
```

Note:

- a** dynlen can be either a compile time constant or a variable with a value that is assigned at run time.
- b** Storage is dynamically allocated for hvcharp.data.
- c** The SQL statement references the name of the structure, not an element within the structure.

Examples of array pointer host variable references:

Table 59. Example references to array pointer host variables

Declaration	Description	Reference
short * hvarrp1[6]	hvarrp1 is an array of 6 pointers that point to two bytes of storage each.	EXEC SQL set:*hvarrp1[n]=123;
double * hvarrp2[3]	hvarrp2 is an array of 3 pointers that point to 8 bytes of storage each.	EXEC SQL set:*hvarrp2[n]=456;
struct { unsigned long len; char * data; } hvbarrp3[5];	hvbarrp3 is an array of 5 bounded character pointers.	EXEC SQL set :hvarrp3[n] = 'data buffer with length'

Example of a structure array host variable reference: Suppose that your program declares the following pointer to the structure tbl_struct:

```
struct tbl_struct *ptr_tbl_struct =
    (struct tbl_struct *) malloc (sizeof (struct tbl_struct) * n);
```

To reference this data in SQL statements, use the pointer as shown in the following example. Assume that tbl_sel_cur is a declared cursor.


```

for (L_col_cnt = 0; L_col_cnt < n; L_col_cnt++)
{
  ...
  EXEC SQL FETCH tbl_sel_cur INTO :ptr_tbl_struct [L_col_cnt]
  ...
}

```

Related tasks:

“Declaring pointer host variables in C programs”

Declaring pointer host variables in C programs

If you use the DB2 coprocessor, you can use pointer host variables with statically or dynamically allocated storage. These pointer host variables can point to numeric data, non-numeric data, or a structure.

You can declare the following types of pointer host variables:

scalar pointer host variable

A host variable that points to numeric or non-numeric scalar data.

array pointer host variable

A host variable that is an array of pointers.

structure array host variable

A host variable that points to a structure.

To declare pointer host variables in C and C++ programs:

Include an asterisk (*) in each variable declaration to indicate that the variable is a pointer.

Restrictions:

- You cannot use pointer host variables that point to character data of an unknown length. For example, do not specify the following declaration: `char * hvcharpu`. Instead, specify the length of the data by using a bounded character pointer host variable. A *bounded character pointer host variable* is a host variable that is declared as a structure with the following elements:
 - A 4-byte field that contains the length of the storage area.
 - A pointer to the non-numeric dynamic storage area.
- You cannot use untyped pointers. For example, do not specify the following declaration: `void * untypedprt`.

Examples of scalar pointer host variable declarations:

Table 60. Example declarations of scalar pointer host variables

Declaration	Description
<code>short *hvshortp;</code>	hvshortp is a pointer host variable that points to two bytes of storage.
<code>double *hvdoubp;</code>	hvdoubp is a pointer host variable that points to eight bytes of storage.
<code>char (*hvcharpn) [20];</code>	hvcharpn is a pointer host variable that points to a nul-terminated character array of up to 20 bytes.

Example of a bounded character pointer host variable declaration: The following example code declares a bounded character pointer host variable called hvbcharp with two elements: len and data.

```

struct {
    unsigned long len;
    char * data;
} hvbcharp;

```

Examples of array pointer host variable declarations:

Table 61. Example declarations of array pointer host variables

Declaration	Description
<code>short * hvarrp1[6]</code>	hvarrp1 is an array of 6 pointers that point to two bytes of storage each.
<code>double * hvarrp2[3]</code>	hvarrp2 is an array of 3 pointers that point to 8 bytes of storage each.
<pre> struct { unsigned long len; char * data; } hvbarrp3[5]; </pre>	hvbarrp3 is an array of 5 bounded character pointers.

Example of a structure array host variable declaration: The following example code declares a table structure called `tbl_struct`.

```

struct tbl_struct
{
    char colname[20];
    small int colno;
    small int coltype;
    small int collen;
};

```

The following example code declares a pointer to the structure `tbl_struct`. Storage is allocated dynamically for up to `n` rows.

```

struct tbl_struct *ptr_tbl_struct =
    (struct tbl_struct *) malloc (sizeof (struct tbl_struct) * n);

```

Related tasks:

“Referencing pointer host variables in C programs” on page 298

Equivalent SQL and C data types

When you declare host variables in your C programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base `SQLTYPE` and `SQLLEN` values that the precompiler uses for host variables in SQL statements.

Table 62. SQL data types, `SQLLEN` values, and `SQLTYPE` values that the precompiler uses for host variables in C programs

C host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
short int	500	2	SMALLINT
long int	496	4	INTEGER
long long long long int sqlint64	492	8	BIGINT ⁵

Table 62. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in C programs (continued)

C host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
decimal(<i>p,s</i>) ²	484	<i>p</i> in byte 1, <i>s</i> in byte 2	DECIMAL(<i>p,s</i>) ²
• <code>_Decimal32</code>	996/997	4	DECFLOAT(16) ^{7, 8}
• <code>_Decimal64</code>	996/997	8	DECFLOAT(16) ⁸
• <code>_Decimal128</code>	996/997	16	DECFLOAT(34) ⁸
float	480	4	FLOAT (single precision)
double	480	8	FLOAT (double precision)
• SQL TYPE IS BINARY(<i>n</i>), 1<= <i>n</i> <=255	912	<i>n</i>	BINARY(<i>n</i>)
• SQL TYPE IS VARBINARY(<i>n</i>), 1<= <i>n</i> <=32704	908	<i>n</i>	VARBINARY(<i>n</i>)
Single-character form	452	1	CHAR(1)
NUL-terminated character form	460	<i>n</i>	VARCHAR (<i>n</i> -1)
VARCHAR structured form 1<= <i>n</i> <=255	448	<i>n</i>	VARCHAR(<i>n</i>)
VARCHAR structured form <i>n</i> >255	456	<i>n</i>	VARCHAR(<i>n</i>)
Single-graphic form	468	1	GRAPHIC(1)
NUL-terminated graphic form	400	<i>n</i>	VARGRAPHIC (<i>n</i> -1)
VARGRAPHIC structured form 1<= <i>n</i> <128	464	<i>n</i>	VARGRAPHIC(<i>n</i>)
VARGRAPHIC structured form <i>n</i> >127	472	<i>n</i>	VARGRAPHIC(<i>n</i>)
• SQL TYPE IS RESULT_SET _LOCATOR	972	4	Result set locator ³
• SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	976	4	Table locator ³

Table 62. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in C programs (continued)

C host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator ³
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator ³
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator ³
SQL TYPE IS BLOB(<i>n</i>) 1≤ <i>n</i> ≤2147483647	404	<i>n</i>	BLOB(<i>n</i>)
SQL TYPE IS CLOB(<i>n</i>) 1≤ <i>n</i> ≤2147483647	408	<i>n</i>	CLOB(<i>n</i>)
SQL TYPE IS DBCLOB(<i>n</i>) 1≤ <i>n</i> ≤1073741823	412	<i>n</i>	DBCLOB(<i>n</i>) ⁴
SQL TYPE IS XML AS BLOB(<i>n</i>)	404	0	XML
SQL TYPE IS XML AS CLOB(<i>n</i>)	408	0	XML
SQL TYPE IS XML AS DBCLOB(<i>n</i>)	412	0	XML
SQL TYPE IS BLOB_FILE	916/917	267	BLOB file reference ³
SQL TYPE IS CLOB_FILE	920/921	267	CLOB file reference ³
SQL TYPE IS DBCLOB_FILE	924/925	267	DBCLOB file reference ³
SQL TYPE IS XML AS BLOB_FILE	916/917	267	XML BLOB file reference ³
SQL TYPE IS XML AS CLOB_FILE	920/921	267	XML CLOB file reference ³
SQL TYPE IS XML AS DBCLOB_FILE	924/925	267	XML DBCLOB file reference ³
SQL TYPE IS ROWID	904	40	ROWID

Table 62. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in C programs (continued)

C host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
Notes:			
1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.			
2. <i>p</i> is the <i>precision</i> ; in SQL terminology, this the total number of digits. In C, this is called the <i>size</i> .			
<i>s</i> is the <i>scale</i> ; in SQL terminology, this is the number of digits to the right of the decimal point. In C, this is called the <i>precision</i> .			
C++ does not support the decimal data type.			
3. Do not use this data type as a column type.			
4. <i>n</i> is the number of double-byte characters.			
5. No exact equivalent. Use DECIMAL(19,0).			
6. The C data type long maps to the SQL data type BIGINT.			
7. DFP host variable with a length of 4 is supported while DFP column can be defined only with length 8(DECFLOAT(16)) or 16(DECFLOAT(34)).			
8. To use the decimal floating-point host data type, you must do the following:			
<ul style="list-style-type: none"> • Use z/OS 1.10 or above (z/OS V1R10 XL C/C++). • Compile with the C/C++ compiler option, DFP. • Specify the SQL compiler option to enable the DB2 coprocessor. • Specify C/C++ compiler option, ARCH(7). It is required by the DFP compiler option if the DFP type is used in the source. • Specify 'DEFINE(__STDC_WANT_DEC_FP__)' compiler option because DFP is not officially part of the C/C++ Language Standard. 			

The following table shows equivalent C host variables for each SQL data type. Use this table to determine the C data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define a variable of NUL-terminated character form or VARCHAR structured form

This table shows direct conversions between SQL data types and C data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 converts those compatible data types.

Table 63. C host variable equivalents that you can use when retrieving data of a particular SQL data type

SQL data type	C host variable equivalent	Notes
SMALLINT	short int	
INTEGER	long int	
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	decimal	You can use the double data type if your C compiler does not have a decimal data type; however, double is not an exact equivalent.
REAL or FLOAT(<i>n</i>)	float	1<= <i>n</i> <=21
DOUBLE PRECISION or FLOAT(<i>n</i>)	double	22<= <i>n</i> <=53
DECFLOAT(16)	_Decminal32	
DECFLOAT(34)	_Decimal128	

Table 63. C host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	C host variable equivalent	Notes
BIGINT	long long, long long int, and sqlint64	
BINARY(<i>n</i>)	SQL TYPE IS BINARY(<i>n</i>)	1<= <i>n</i> <=255 If data can contain character NULs (\0), certain C and C++ library functions might not handle the data correctly. Ensure that your application handles the data properly.
VARBINARY(<i>n</i>)	SQL TYPE IS VARBINARY(<i>n</i>)	1<= <i>n</i> <=32 704
CHAR(1)	single-character form	
CHAR(<i>n</i>)	no exact equivalent	If <i>n</i> >1, use NUL-terminated character form
VARCHAR(<i>n</i>)	NUL-terminated character form VARCHAR structured form	If data can contain character NULs (\0), use VARCHAR structured form. Allow at least <i>n</i> +1 to accommodate the NUL-terminator.
GRAPHIC(1)	single-graphic form	
GRAPHIC(<i>n</i>)	no exact equivalent	If <i>n</i> >1, use NUL-terminated graphic form. <i>n</i> is the number of double-byte characters.
VARGRAPHIC(<i>n</i>)	NUL-terminated graphic form VARGRAPHIC structured form	If data can contain graphic NUL values (\0\0), use VARGRAPHIC structured form. Allow at least <i>n</i> +1 to accommodate the NUL-terminator. <i>n</i> is the number of double-byte characters.
DATE	NUL-terminated character form VARCHAR structured form	If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 11 characters to accommodate the NUL-terminator. If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 10 characters.
TIME	NUL-terminated character form VARCHAR structured form	If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 7; to include seconds, the length must be at least 9 to accommodate the NUL-terminator. If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 6; to include seconds, the length must be at least 8.

Table 63. C host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	C host variable equivalent	Notes
TIMESTAMP	NUL-terminated character form	The length must be at least 20. To include microseconds, the length must be 27. If the length is less than 27, truncation occurs on the microseconds part.
	VARCHAR structured form	The length must be at least 19. To include microseconds, the length must be 26. If the length is less than 26, truncation occurs on the microseconds part.
TIMESTAMP(0)	NUL-terminated character form	The length must be at least 20.
	VARCHAR structured form	The length must be at least 19.
TIMESTAMP(<i>p</i>) <i>p</i> > 0	NUL-terminated character form	The length must be at least 20. To include fractional seconds, the length must be 21+ <i>x</i> where <i>x</i> is the number of fractional seconds to include; if <i>x</i> is less than <i>p</i> , truncation occurs on the fraction seconds part.
	VARCHAR structured form	The length must be at least 19. To include fractional seconds, the length must be 20+ <i>x</i> where <i>x</i> is the number of fractional seconds to include; if <i>x</i> is less than <i>p</i> , truncation occurs on the fractional seconds part.
TIMESTAMP(0) WITH TIME ZONE	NUL-terminated character form	The length must be at least 26.
	VARCHAR structured form	The length must be at least 25.
TIMESTAMP(<i>p</i>) WITH TIME ZONE	NUL-terminated character form	The length must be at least 27+ <i>p</i> .
	VARCHAR structured form	The length must be at least 26+ <i>p</i> .
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB(<i>n</i>)	SQL TYPE IS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
CLOB(<i>n</i>)	SQL TYPE IS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
DBCLOB(<i>n</i>)	SQL TYPE IS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823
XML	SQL TYPE IS XML AS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647

Table 63. C host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	C host variable equivalent	Notes
XML	SQL TYPE IS XML AS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823
BLOB file reference	SQL TYPE IS BLOB_FILE	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB file reference	SQL TYPE IS CLOB_FILE	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB file reference	SQL TYPE IS DBCLOB_FILE	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
XML BLOB file reference	SQL TYPE IS XML AS BLOB_FILE	Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.
XML CLOB file reference	SQL TYPE IS XML AS CLOB_FILE	Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.
XML DBCLOB file reference	SQL TYPE IS XML AS DBCLOB_FILE	Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.
ROWID	SQL TYPE IS ROWID	

Related concepts:

“Compatibility of SQL and language data types” on page 180

“LOB host variable, LOB locator, and LOB file reference variable declarations” on page 757

“Host variable data types for XML data in embedded SQL applications” on page 241

SQL statements in C programs

You can code SQL statements in a C program wherever you can use executable statements.

Each SQL statement in a C program must begin with EXEC SQL and end with a semicolon (;). The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

In general, because C is case sensitive, use uppercase letters to enter all SQL keywords. However, if you use the FOLD precompiler suboption, DB2 folds lowercase letters in SBCS SQL ordinary identifiers to uppercase. For information about host language precompiler options, see Table 150 on page 933.

You must keep the case of host variable names consistent throughout the program. For example, if a host variable name is lowercase in its declaration, it must be lowercase in all SQL statements. You might code an UPDATE statement in a C program as follows:

```
EXEC SQL
UPDATE DSN8A10.DEPT
SET MGRNO = :mgr_num
WHERE DEPTNO = :int_dept;
```

Comments: You can include C comments (`/* ... */`) within SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can use single-line comments (starting with `//`) in C language statements, but not in embedded SQL. You can use SQL comments within embedded SQL statements. You can nest comments.

To include EBCDIC DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string.

Continuation for SQL statements: You can use a backslash to continue a character-string constant or delimited identifier on the following line. However, EBCDIC DBCS string constants cannot be continued on a second line.

Declaring tables and views: Your C program should use the DECLARE TABLE statement to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. For more information, see "DCLGEN (declarations generator)" on page 161.

Including SQL statements and variable declarations in source code that is to be processed by the DB2 precompiler: To include SQL statements or C host variable declarations from a member of a partitioned data set, add the following SQL statement to the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use C `#include` statements to include SQL statements or C host variable declarations.

Margins: Code SQL statements in columns 1 through 72, unless you specify other margins to the DB2 precompiler. If EXEC SQL is not within the specified margins, the DB2 precompiler does not recognize the SQL statement. The margin rules do not apply to the DB2 coprocessor. The DB2 coprocessor allows variable length source input.

Names: You can use any valid C name for a host variable, subject to the following restrictions:

- Do not use DBCS characters.
- Do not use external entry names or access plan names that begin with 'DSN', and do not use host variable names or macro names that begin with 'SQL' (in any combination of uppercase or lowercase letters). These names are reserved for DB2.

Nulls and NULs: C and SQL differ in the way they use the word *null*. 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 that 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 non-null values and denotes the absence of a (nonnull) value. NUL (or NUL-terminator) is the null character in C and C++, and NULL is the SQL null value.

Sequence numbers: The DB2 precompiler generates statements without sequence numbers. (The DB2 coprocessor does not perform this action, because the source is read and modified by the compiler.)

Statement labels: You can precede SQL statements with a label.

Trigraph characters: Some characters from the C character set are not available on all keyboards. You can enter these characters into a C source program using a sequence of three characters called a *trigraph*. The trigraph characters that DB2 supports are the same as those that the C compiler supports.

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be within the scope of any SQL statements that the statement WHENEVER affects.

Special C/C++ considerations:

- Using the C/370™ multi-tasking facility, in which multiple tasks execute SQL statements, causes unpredictable results.
- Except for the DB2 coprocessor, you must run the DB2 precompiler before running the C preprocessor.
- Except for the DB2 coprocessor, DB2 precompiler does not support C preprocessor directives.
- If you use conditional compiler directives that contain C code, either place them after the first C token in your application program, or include them in the C program using the #include preprocessor directive.

Refer to the appropriate C documentation for more information about C preprocessor directives.

To use the decimal floating-point host data type, you must do the following:

- Use z/OS 1.10 or above (z/OS V1R10 XL C/C++).
- Compile with the C/C++ compiler option, DFP.
- Specify the SQL compiler option to enable the DB2 coprocessor.
- Specify C/C++ compiler option, ARCH(7). It is required by the DFP compiler option if the DFP type is used in the source.
- Specify 'DEFINE(__STDC_WANT_DEC_FP__)' compiler option.

Handling SQL error return codes in C or C++

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information about the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 229.

You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234.

DSNTIAR syntax:

```
rc = DSNTIAR(&sqlca, &message, &lrecl);
```

The DSNTIAR parameters have the following meanings:

&sqlca

An SQL communication area.

&message

An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *&lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
#define data_len 132
#define data_dim 10
int length_of_line = data_len ;
struct error_struct {
    short int error_len;
    char error_text[data_dim][data_len];
} error_message = {data_dim * data_len};
:
rc = DSNTIAR(&sqlca, &error_message, &length_of_line);
```

where *error_message* is the name of the message output area, *data_dim* is the number of lines in the message output area, and *data_len* is the length of each line.

&lrecl

A fullword containing the logical record length of output messages, between 72 and 240.

To inform your compiler that DSNTIAR is an assembler language program, include one of the following statements in your application.

For C, include:

```
#pragma linkage (DSNTIAR,OS)
```

For C++, include a statement similar to this:

```
extern "OS" short int DSNTIAR(struct sqlca *sqlca,
                             struct error_struct *error_message,
                             int *data_len);
```

Examples of calling DSNTIAR from an application appear in the DB2 sample C program DSN8BD3 and in the sample C++ program DSN8BE3. Both are in the library DSN8A10.SDSNSAMP. See "DB2 sample applications" on page 1092 for instructions on how to access and print the source code for the sample programs.

CICS: If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
rc = DSNTIAC(&eib, &commarea, &sqlca, &message, &lrecl);
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

&eib EXEC interface block

&commarea

communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix.SDSNSAMP*.

Related concepts:


“Host variable arrays in an SQL statement” on page 191

Related tasks:

“Including dynamic SQL in your program” on page 193

“Embedding SQL statements in your application” on page 183

“Handling SQL error codes” on page 239

 Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Delimiters in SQL statements in C programs

You must delimit SQL statements in your C program so that DB2 knows when a particular SQL statement ends.


Delimit an SQL statement in your C program with the beginning keyword EXEC SQL and a Semicolon (;).

Programming examples in C

You can write DB2 programs in C and C++. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, start with the JCL in member DSNTJ2D of data set *prefix.SDSNSAMP* as a model for your JCL. *prefix* is the high-order qualifier for the data set that contains the sample jobs after they are customized by the installation process.

Related concepts:

 Job DSNTJ2D (DB2 Installation and Migration)

Related reference:

“Programming examples” on page 251

Sample dynamic and static SQL in a C program

Programs that access DB2 can contain static SQL, dynamic SQL, or both.

This example shows a C program that contains both static and dynamic SQL.

The following figure illustrates dynamic SQL and static SQL embedded in a C program. Each section of the program is identified with a comment. Section 1 of the program shows static SQL; sections 2, 3, and 4 show dynamic SQL. The function of each section is explained in detail in the prologue to the program.

```

/*****
/* Descriptive name = Dynamic SQL sample using C language */
/* */
/* Function = To show examples of the use of dynamic and static */
/* SQL. */
/* */
/* Notes = This example assumes that the EMP and DEPT tables are */
/* defined. They need not be the same as the DB2 Sample */
/* tables. */
/* */
/* Module type = C program */
/* Processor = DB2 precompiler, C compiler */
/* Module size = see link edit */
/* Attributes = not reentrant or reusable */
/* */
/* Input = */
/* */
/* symbolic label/name = DEPT */
/* description = arbitrary table */
/* symbolic label/name = EMP */
/* description = arbitrary table */
/* */
/* Output = */
/* */
/* symbolic label/name = SYSPRINT */
/* description = print results via printf */
/* */
/* Exit-normal = return code 0 normal completion */
/* */
/* Exit-error = */
/* */
/* Return code = SQLCA */
/* */
/* Abend codes = none */
/* */
/* External references = none */
/* */
/* Control-blocks = */
/* SQLCA - sql communication area */
/* */
/* Logic specification: */
/* */
/* There are four SQL sections. */
/* */
/* 1) STATIC SQL 1: using static cursor with a SELECT statement. */
/* Two output host variables. */
/* 2) Dynamic SQL 2: Fixed-list SELECT, using same SELECT statement */
/* used in SQL 1 to show the difference. The prepared string */
/* :iptstr can be assigned with other dynamic-able SQL statements.*/
/* 3) Dynamic SQL 3: Insert with parameter markers. */
/* Using four parameter markers which represent four input host */
/* variables within a host structure. */
/* 4) Dynamic SQL 4: EXECUTE IMMEDIATE */
/* A GRANT statement is executed immediately by passing it to DB2 */
/* via a varying string host variable. The example shows how to */
/* set up the host variable before passing it. */
/* */
/*****

#include "stdio.h"
#include "stdefs.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL BEGIN DECLARE SECTION;
short edlevel;
struct { short len;
char x1??(56??);

```

```

    } stmbf1, stmbf2, inpstr;
struct { short len;
        char x1??(15??);
    } lname;
short hv1;
struct { char deptno??(4??);
        struct { short len;
                char x??(36??);
            } deptname;
        char mgrno??(7??);
        char admrdept??(4??);
        char location??(17??);
    } hv2;
short ind??(4??);
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE EMP TABLE
    (EMPNO          CHAR(6)          ,
     FIRSTNAME     VARCHAR(12)       ,
     MIDINIT       CHAR(1)          ,
     LASTNAME      VARCHAR(15)       ,
     WORKDEPT      CHAR(3)          ,
     PHONENO       CHAR(4)          ,
     HIREDATE      DECIMAL(6)        ,
     JOBCODE       DECIMAL(3)        ,
     EDLEVEL       SMALLINT         ,
     SEX           CHAR(1)          ,
     BIRTHDATE     DECIMAL(6)        ,
     SALARY        DECIMAL(8,2)     ,
     FORFNAME      VARGRAPHIC(12)   ,
     FORMNAME      GRAPHIC(1)       ,
     FORLNAME      VARGRAPHIC(15)   ,
     FORADDR       VARGRAPHIC(256)  ) ;
EXEC SQL DECLARE DEPT TABLE
    (
     DEPTNO        CHAR(3)          ,
     DEPTNAME      VARCHAR(36)      ,
     MGRNO         CHAR(6)          ,
     ADMRDEPT      CHAR(3)          ,
     LOCATION      CHAR(16));

main ()
{
printf("??/n***      begin of program                ***");
EXEC SQL WHENEVER SQLERROR GO TO HANDLERR;
EXEC SQL WHENEVER SQLWARNING GO TO HANDWARN;
EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND;
/*****
/* Assign values to host variables which will be input to DB2 */
*****/
strcpy(hv2.deptno,"M92");
strcpy(hv2.deptname.x,"DDL");
hv2.deptname.len = strlen(hv2.deptname.x);
strcpy(hv2.mgrno,"000010");
strcpy(hv2.admrdept,"A00");
/*****
/* Static SQL 1: DECLARE CURSOR, OPEN, FETCH, CLOSE */
/* Select into :edlevel, :lname */
*****/
printf("??/n***      begin declare                ***");
EXEC SQL DECLARE C1 CURSOR FOR SELECT EDLEVEL, LASTNAME FROM EMP
        WHERE EMPNO = '000010';
printf("??/n***      begin open                    ***");
EXEC SQL OPEN C1;

printf("??/n***      begin fetch                    ***");
EXEC SQL FETCH C1 INTO :edlevel, :lname;
printf("??/n***      returned values                ***");
printf("??/n??/nedlevel = %d",edlevel);

```



```

printf("??/nlname = %s\n",lname.x1);

printf("??/n***      begin close          ***");
EXEC SQL CLOSE C1;
/*****
/* Dynamic SQL 2:  PREPARE, DECLARE CURSOR, OPEN, FETCH, CLOSE */
/* Select into :edlevel, :lname */
*****/
sprintf (inpstr.x1,
        "SELECT EDLEVEL, LASTNAME FROM EMP WHERE EMPNO = '000010'");
inpstr.len = strlen(inpstr.x1);
printf("??/n***      begin prepare        ***");
EXEC SQL PREPARE STAT1 FROM :inpstr;
printf("??/n***      begin declare       ***");
EXEC SQL DECLARE C2 CURSOR FOR STAT1;
printf("??/n***      begin open          ***");
EXEC SQL OPEN C2;

printf("??/n***      begin fetch          ***");
EXEC SQL FETCH C2 INTO :edlevel, :lname;
printf("??/n***      returned values     ***");
printf("??/n??/nedlevel = %d",edlevel);
printf("??/nlname = %s??/n",lname.x1);

printf("??/n***      begin close          ***");
EXEC SQL CLOSE C2;
/*****
/* Dynamic SQL 3:  PREPARE with parameter markers */
/* Insert into with five values. */
*****/
sprintf (stmbf1.x1,
        "INSERT INTO DEPT VALUES (?, ?, ?, ?, ?)");
stmbf1.len = strlen(stmbf1.x1);
printf("??/n***      begin prepare        ***");
EXEC SQL PREPARE s1 FROM :stmbf1;
printf("??/n***      begin execute       ***");
EXEC SQL EXECUTE s1 USING :hv2:ind;
printf("??/n***      following are expected insert results ***");
printf("??/n hv2.deptno = %s",hv2.deptno);
printf("??/n hv2.deptname.len = %d",hv2.deptname.len);
printf("??/n hv2.deptname.x = %s",hv2.deptname.x);
printf("??/n hv2.mgrno = %s",hv2.mgrno);
printf("??/n hv2.admrdept = %s",hv2.admrdept);
printf("??/n hv2.location = %s",hv2.location);
EXEC SQL COMMIT;
/*****
/* Dynamic SQL 4:  EXECUTE IMMEDIATE */
/* Grant select */
*****/
sprintf (stmbf2.x1,
        "GRANT SELECT ON EMP TO USERX");
stmbf2.len = strlen(stmbf2.x1);
printf("??/n***      begin execute immediate ***");
EXEC SQL EXECUTE IMMEDIATE :stmbf2;
printf("??/n***      end of program      ***");
goto progend;
HANDWARN: HANDLERR: NOTFOUND: ;
printf("??/n SQLCODE = %d",SQLCODE);
printf("??/n SQLWARN0 = %c",SQLWARN0);
printf("??/n SQLWARN1 = %c",SQLWARN1);
printf("??/n SQLWARN2 = %c",SQLWARN2);
printf("??/n SQLWARN3 = %c",SQLWARN3);
printf("??/n SQLWARN4 = %c",SQLWARN4);
printf("??/n SQLWARN5 = %c",SQLWARN5);
printf("??/n SQLWARN6 = %c",SQLWARN6);

```

```

printf("??/n SQLWARN7 = %c",SQLWARN7);
printf("??/n SQLERRMC = %s",sqlca.sqlerrmc);
progend: ;
}

```

Example C program that calls a stored procedure

You can call the C language version of the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention.

Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets. The following figure contains the example C program that calls the GETPRML stored procedure.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
    /******
    /* Include the SQLCA and SQLDA
    /******
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL INCLUDE SQLDA;
    /******
    /* Declare variables that are not SQL-related.
    /******
    short int i;          /* Loop counter
    /******
    /* Declare the following:
    /* - Parameters used to call stored procedure GETPRML
    /* - An SQLDA for DESCRIBE PROCEDURE
    /* - An SQLDA for DESCRIBE CURSOR
    /* - Result set variable locators for up to three result
    /* sets
    /******
    EXEC SQL BEGIN DECLARE SECTION;
    char procnm[19];      /* INPUT parm -- PROCEDURE name */
    char schema[9];      /* INPUT parm -- User's schema */
    long int out_code;    /* OUTPUT -- SQLCODE from the
    /* SELECT operation.
    /*
    struct {
        short int parmlen;
        char parmtxt[254];
    } parmlst;          /* OUTPUT -- RUNOPTS values
    /* for the matching row in
    /* catalog table SYSROUTINES
    /*
    struct indicators {
        short int procnm_ind;
        short int schema_ind;
        short int out_code_ind;
        short int parmlst_ind;
    } parmind;          /* Indicator variable structure */
    struct sqlda *proc_da;
    /* SQLDA for DESCRIBE PROCEDURE
    /*
    struct sqlda *res_da;
    /* SQLDA for DESCRIBE CURSOR
    /*
    static volatile
    SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
    /* Locator variables
    /*
    EXEC SQL END DECLARE SECTION;

    /******
    /* Allocate the SQLDAs to be used for DESCRIBE
    /*
    /* PROCEDURE and DESCRIBE CURSOR. Assume that at most
    /*

```

```

/* three cursors are returned and that each result set      */
/* has no more than five columns.                          */
/*****
proc_da = (struct sqlda *)malloc(SQLDASIZE(3));
res_da = (struct sqlda *)malloc(SQLDASIZE(5));

/*****
/* Call the GETPRML stored procedure to retrieve the      */
/* RUNOPTS values for the stored procedure.  In this     */
/* example, we request the PARMLIST definition for the    */
/* stored procedure named DSN8EP2.                       */
/*                                                       */
/* The call should complete with SQLCODE +466 because    */
/* GETPRML returns result sets.                         */
/*****
strcpy(procnm,"dsn8ep2");
/* Input parameter -- PROCEDURE to be found */
strcpy(schema,"");
/* Input parameter -- Schema name for proc */
parmind.procnm_ind=0;
parmind.schema_ind=0;
parmind.out_code_ind=0;
/* Indicate that none of the input parameters */
/* have null values */
parmind.parmlst_ind=-1;
/* The parmlst parameter is an output parm. */
/* Mark PARMLST parameter as null, so the DB2 */
/* requester does not have to send the entire */
/* PARMLST variable to the server. This */
/* helps reduce network I/O time, because */
/* PARMLST is fairly large. */
EXEC SQL
CALL GETPRML(:procnm INDICATOR :parmind.procnm_ind,
             :schema INDICATOR :parmind.schema_ind,
             :out_code INDICATOR :parmind.out_code_ind,
             :parmlst INDICATOR :parmind.parmlst_ind);
if(SQLCODE!=+466) /* If SQL CALL failed, */
{
/* print the SQLCODE and any */
/* message tokens */
printf("SQL CALL failed due to SQLCODE =
printf("sqlca.sqlerrmc = ");
for(i=0;i<sqlca.sqlerrml;i++)
printf("i]);
printf("\n");
}
else /* If the CALL worked, */
if(out_code!=0) /* Did GETPRML hit an error? */
printf("GETPRML failed due to RC =
/*****
/* If everything worked, do the following: */
/* - Print out the parameters returned. */
/* - Retrieve the result sets returned. */
/*****
else
{
printf("RUNOPTS =
/* Print out the runopts list */

/*****
/* Use the statement DESCRIBE PROCEDURE to */
/* return information about the result sets in the */
/* SQLDA pointed to by proc_da: */
/* - SQLD contains the number of result sets that were */
/* returned by the stored procedure. */
/* - Each SQLVAR entry has the following information */
/* about a result set: */

```

```

/* - SQLNAME contains the name of the cursor that */
/* the stored procedure uses to return the result */
/* set. */
/* - SQLIND contains an estimate of the number of */
/* rows in the result set. */
/* - SQLDATA contains the result locator value for */
/* the result set. */
/*****
EXEC SQL DESCRIBE PROCEDURE INTO :*proc_da;
/*****
/* Assume that you have examined SQLD and determined */
/* that there is one result set. Use the statement */
/* ASSOCIATE LOCATORS to establish a result set locator */
/* for the result set. */
/*****
EXEC SQL ASSOCIATE LOCATORS (:loc1) WITH PROCEDURE GETPRML;

/*****
/* Use the statement ALLOCATE CURSOR to associate a */
/* cursor for the result set. */
/*****
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
/*****
/* Use the statement DESCRIBE CURSOR to determine the */
/* columns in the result set. */
/*****
EXEC SQL DESCRIBE CURSOR C1 INTO :*res_da;

/*****
/* Call a routine (not shown here) to do the following: */
/* - Allocate a buffer for data and indicator values */
/* fetched from the result table. */
/* - Update the SQLDATA and SQLIND fields in each */
/* SQLVAR of *res_da with the addresses at which to */
/* to put the fetched data and values of indicator */
/* variables. */
/*****
alloc_outbuff(res_da);

/*****
/* Fetch the data from the result table. */
/*****
while(SQLCODE==0)
    EXEC SQL FETCH C1 USING DESCRIPTOR :*res_da;
}
return;
}

```

Example C stored procedure with a GENERAL linkage convention

You can call a stored procedure that uses the GENERAL linkage convention from a C program.

This example stored procedure does the following:

- Searches the DB2 catalog table SYSROUTINES for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention used for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT statement and the value of the RUNOPTS column from SYSROUTINES.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
    OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
    LANGUAGE C
    DETERMINISTIC
    READS SQL DATA
    EXTERNAL NAME "GETPRML"
    COLLID GETPRML
    ASUTIME NO LIMIT
    PARAMETER STYLE GENERAL
    STAY RESIDENT NO
    RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
    WLM ENVIRONMENT SAMPPROG
    PROGRAM TYPE MAIN
    SECURITY DB2
    RESULT SETS 2
    COMMIT ON RETURN NO;
```

The following example is a C stored procedure with linkage convention GENERAL

```
#pragma runopts(plist(os))
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;

/*****
/* Declare C variables for SQL operations on the parameters. */
/* These are local variables to the C program, which you must */
/* copy to and from the parameter list provided to the stored */
/* procedure. */
*****/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char SCHEMA[9];
char PARMLST[255];
EXEC SQL END DECLARE SECTION;

/*****
/* Declare cursors for returning result sets to the caller. */
*****/
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
    SELECT NAME
    FROM SYSIBM.SYSTABLES
    WHERE CREATOR=:SCHEMA;

main(argc,argv)
    int argc;
    char *argv[];
{
    /*****
    /* Copy the input parameters into the area reserved in */
    /* the program for SQL processing. */
    *****/
    strcpy(PROCNM, argv[1]);
    strcpy(SCHEMA, argv[2]);

    /*****
    /* Issue the SQL SELECT against the SYSROUTINES */
    /* DB2 catalog table. */
    *****/
    strcpy(PARMLST, ""); /* Clear PARMLST */
    EXEC SQL
```

```

SELECT RUNOPTS INTO :PARMLST
  FROM SYSIBM.ROUTINES
  WHERE NAME=:PROCNM AND
        SCHEMA=:SCHEMA;

/*****
/* Copy SQLCODE to the output parameter list.      */
*****/
*(int *) argv[3] = SQLCODE;

/*****
/* Copy the PARMLST value returned by the SELECT back to*/
/* the parameter list provided to this stored procedure.*/
*****/
strcpy(argv[4], PARMLST);

/*****
/* Open cursor C1 to cause DB2 to return a result set  */
/* to the caller.                                     */
*****/
EXEC SQL OPEN C1;
}

```

Example C stored procedure with a GENERAL WITH NULLS linkage convention

You can call a stored procedure that uses the GENERAL WITH NULLS linkage convention from a C program.

This example stored procedure does the following:

- Searches the DB2 catalog table SYSROUTINES for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```

CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 2
COMMIT ON RETURN NO;

```

The following example is a C stored procedure with linkage convention GENERAL WITH NULLS.

```
#pragma runopts(plist(os))
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;

/*****
/* Declare C variables used for SQL operations on the
/* parameters. These are local variables to the C program,
/* which you must copy to and from the parameter list provided
/* to the stored procedure.
*****/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char SCHEMA[9];
char PARMLST[255];
struct INDICATORS {
    short int PROCNM_IND;
    short int SCHEMA_IND;
    short int OUT_CODE_IND;
    short int PARMLST_IND;
} PARM_IND;
EXEC SQL END DECLARE SECTION;

/*****
/* Declare cursors for returning result sets to the caller.
*****/
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
    SELECT NAME
    FROM SYSIBM.SYSTABLES
    WHERE CREATOR=:SCHEMA;

main(argc,argv)
int argc;
char *argv[];
{

    /*****
    /* Copy the input parameters into the area reserved in
    /* the local program for SQL processing.
    *****/
    strcpy(PROCNM, argv[1]);
    strcpy(SCHEMA, argv[2]);

    /*****
    /* Copy null indicator values for the parameter list.
    *****/
    memcpy(&PARM_IND,(struct INDICATORS *) argv[5],
        sizeof(PARM_IND));

    /*****
    /* If any input parameter is NULL, return an error
    /* return code and assign a NULL value to PARMLST.
    *****/
    if (PARM_IND.PROCNM_IND<0 ||
        PARM_IND.SCHEMA_IND<0) {
        *(int *) argv[3] = 9999; /* set output return code */
        PARM_IND.OUT_CODE_IND = 0; /* value is not NULL */
        PARM_IND.PARMLST_IND = -1; /* PARMLST is NULL */
    }

    else {
        /*****
        /* If the input parameters are not NULL, issue the SQL
        /* SELECT against the SYSIBM.SYSROUTINES catalog
        /* table.
        *****/

```



```

/*****
strcpy(PARMLST, "");          /* Clear PARMLST          */
EXEC SQL
    SELECT RUNOPTS INTO :PARMLST
          FROM SYSIBM.SYSROUTINES
          WHERE NAME=:PROCNM AND
                SCHEMA=:SCHEMA;
/*****
/* Copy SQLCODE to the output parameter list.          */
/*****
*(int *) argv[3] = SQLCODE;
PARM_IND.OUT_CODE_IND = 0;          /* OUT_CODE is not NULL */
}

/*****
/* Copy the RUNOPTS value back to the output parameter */
/* area.          */
/*****
strcpy(argv[4], PARMLST);

/*****
/* Copy the null indicators back to the output parameter*/
/* area.          */
/*****
memcpy((struct INDICATORS *) argv[5], &PARM_IND,
       sizeof(PARM_IND));

/*****
/* Open cursor C1 to cause DB2 to return a result set  */
/* to the caller.          */
/*****
EXEC SQL OPEN C1;
}

```

Chapter 6. Coding SQL statements in COBOL application programs

When you code SQL statements in COBOL application programs, you should follow certain guidelines.

Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL

COBOL programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

For COBOL programs, when you specify STDSQL(YES), you must declare an SQLCODE variable. DB2 declares an SQLCA area for you in the WORKING-STORAGE SECTION. DB2 controls the structure and location of the SQLCA.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

To define the SQL communications area, SQLSTATE, and SQLCODE:

Choose one of the following actions:

Option	Description
To define the SQL communications area:	<ol style="list-style-type: none">1. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration: <pre>EXEC SQL INCLUDE SQLCA</pre><p>You can specify INCLUDE SQLCA or a declaration for SQLCODE wherever you can specify a 77 level or a record description entry in the WORKING-STORAGE SECTION.</p><p>DB2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.</p>

Option	Description
<p>To declare SQLCODE and SQLSTATE host variables:</p>	<ol style="list-style-type: none"> <li data-bbox="933 226 1414 428">1. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as PIC S9(9) BINARY, PIC S9(9) COMP-4, PIC S9(9) COMP-5, or PICTURE S9(9) COMP. When you use the DB2 precompiler, you can declare a stand-alone SQLCODE variable in either the WORKING-STORAGE SECTION or LINKAGE SECTION. When you use the DB2 coprocessor, you can declare a stand-alone SQLCODE variable in the WORKING-STORAGE SECTION, LINKAGE SECTION or LOCAL-STORAGE SECTION. <li data-bbox="933 737 1414 877">2. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as PICTURE X(5). <p data-bbox="933 898 1414 953">Restriction: Do not declare an SQLSTATE variable as an element of a structure.</p> <p data-bbox="933 957 1414 1098">Requirement: After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</p>

Related tasks:

“Checking the execution of SQL statements” on page 227

“Checking the execution of SQL statements by using the SQLCA” on page 228

“Checking the execution of SQL statements by using SQLCODE and SQLSTATE” on page 232

“Defining the items that your program can use to check whether an SQL statement executed successfully” on page 173

Defining SQL descriptor areas in COBOL

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

To define SQL descriptor areas:

Perform one of the following actions:

- Code the SQLDA declarations directly in your program. When you use the DB2 precompiler, you must place SQLDA declarations in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program, wherever you can specify a record description entry in that section. When you use the DB2 coprocessor, you must place SQLDA declarations in the WORKING-STORAGE SECTION,

LINKAGE SECTION or LOCAL-STORAGE SECTION of your program, wherever you can specify a record description entry in that section.

- Call a subroutine that is written in C, PL/I, or assembler language and that uses the INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions that you need.

Restrictions:

- You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.
- You cannot use the SQL INCLUDE statement for the SQLDA, because it is not supported in COBOL.

Related tasks:

“Defining SQL descriptor areas” on page 173

Declaring host variables and indicator variables in COBOL

You can use host variables, host variable arrays, and host structures in SQL statements in your program to pass data between DB2 and your application.

To declare host variables, host variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:
 - You must explicitly declare all host variables and host variable arrays that are used in SQL statements in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program's DATA DIVISION.
 - You must explicitly declare each host variable and host variable array before using them in an SQL statement.
 - You can specify OCCURS when defining an indicator structure, a host variable array, or an indicator variable array. You cannot specify OCCURS for any other type of host variable.
 - You cannot implicitly declare any host variables through default typing or by using the IMPLICIT statement.
 - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
 - If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
 - Ensure that any SQL statement that uses a host variable or host variable array is within the scope of the statement that declares that variable or array.
 - If you are using the DB2 precompiler, ensure that the names of host variables and host variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.
2. Optional: Define any associated indicator variables, arrays, and structures.

Related tasks:

“Declaring host variables and indicator variables” on page 174

Host variables in COBOL

In COBOL programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set and table locators and LOB and XML file reference variables.

Restrictions:

- Only some of the valid COBOL declarations are valid host variable declarations. If the declaration for a variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- You can not use locators as column types.
The following locator data types are COBOL data types and SQL data types:
 - Result set locator
 - Table locator
 - LOB locators
 - LOB file reference variables
- One or more REDEFINES entries can follow any level 77 data description entry. However, you cannot use the names in these entries in SQL statements. Entries with the name FILLER are ignored.

Recommendations:

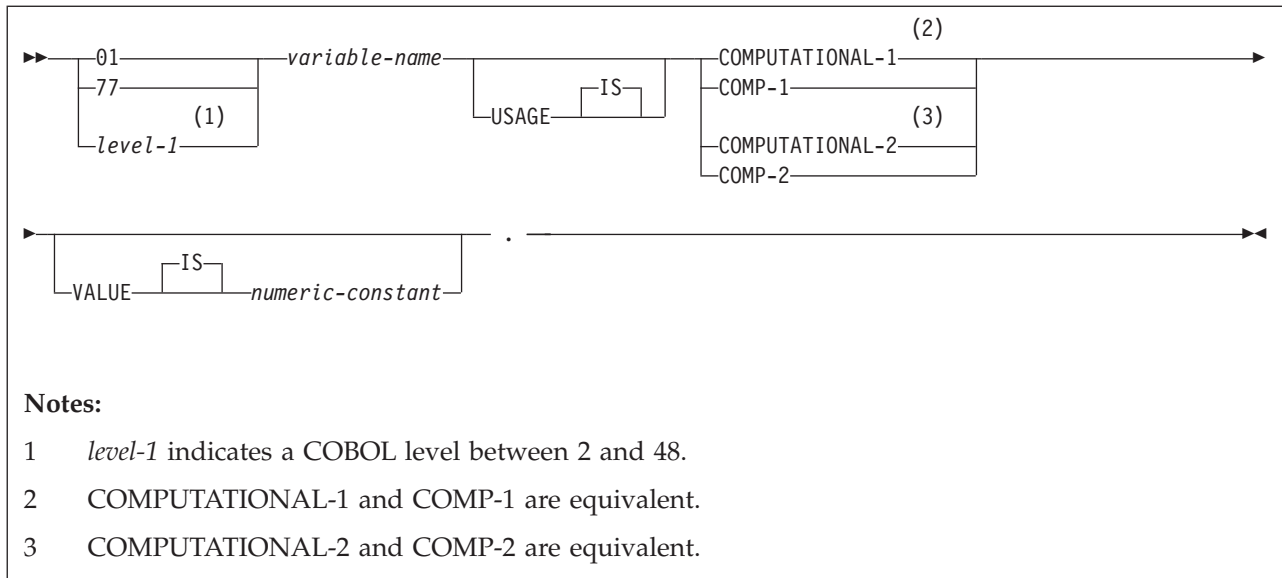
- Be careful of overflow. For example, suppose that you retrieve an INTEGER column value into a PICTURE S9(4) host variable and the column value is larger than 32767 or smaller than -32768. You get an overflow warning or an error, depending on whether you specify an indicator variable.
- Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a PICTURE X(70) host variable, the rightmost 10 characters of the retrieved string are truncated. Retrieving a double precision floating-point or decimal column value into a PIC S9(8) COMP host variable removes any fractional part of the value. Similarly, retrieving a column value with DECIMAL data type into a COBOL decimal variable with a lower precision might truncate the value.
- If your varying-length string host variables receive values whose length is greater than 9999 bytes, compile the applications in which you use those host variables with the option TRUNC(BIN). TRUNC(BIN) lets the length field for the string receive a value of up to 32767 bytes.

Numeric host variables

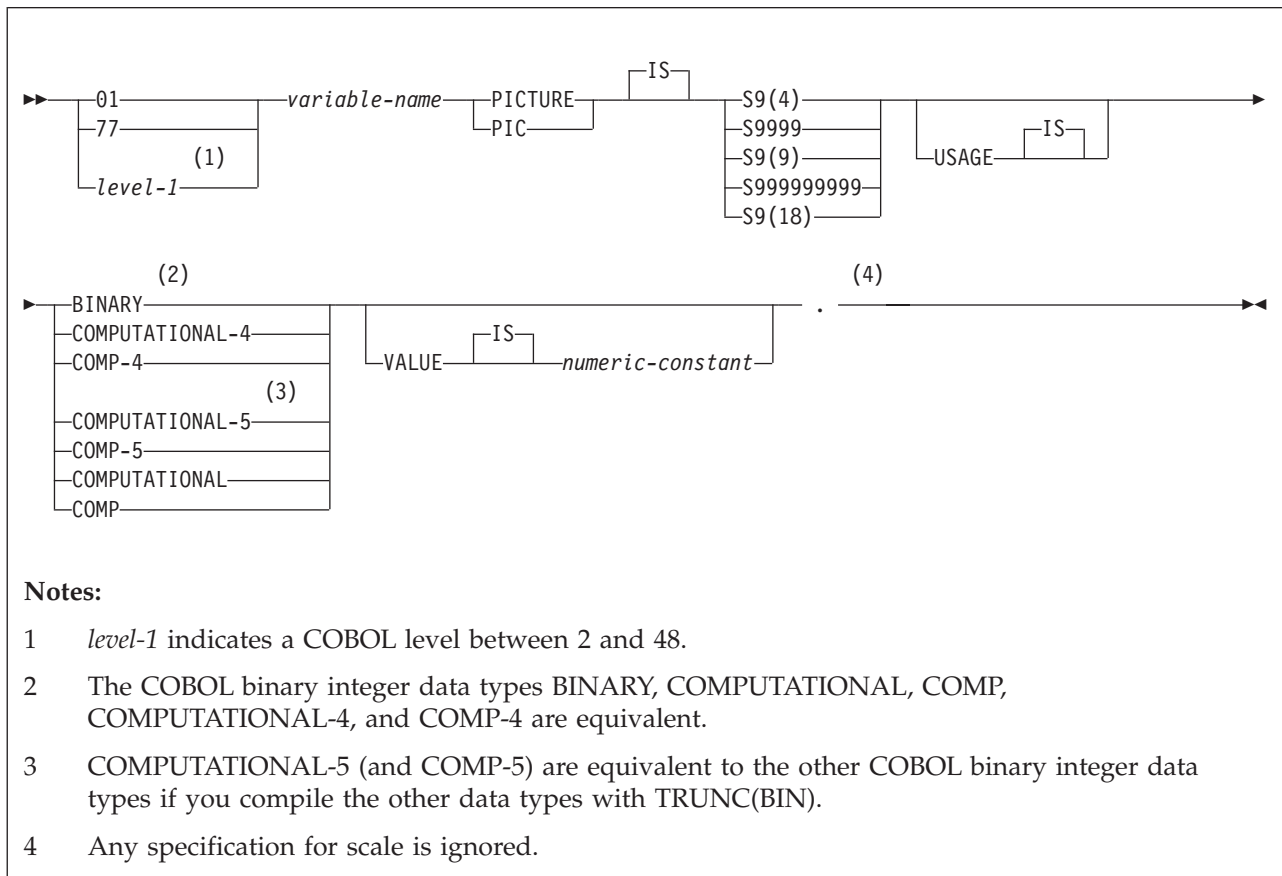
You can specify the following forms of numeric host variables:

- Floating-point numbers
- Integers and small integers
- Decimal numbers

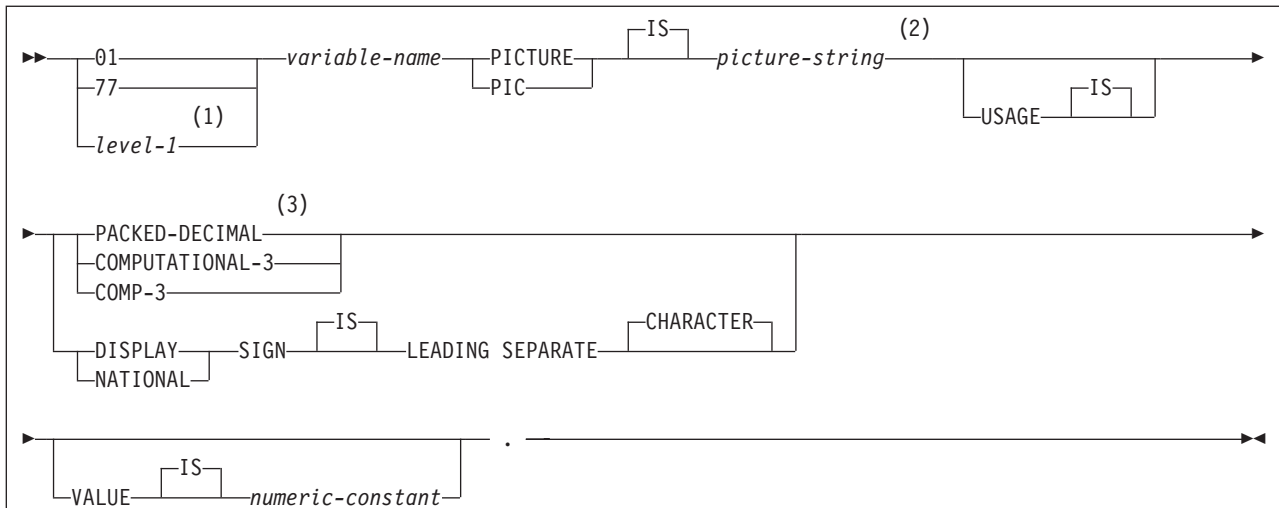
The following diagram shows the syntax for declaring floating-point or real host variables.



The following diagram shows the syntax for declaring integer and small integer host variables.



The following diagram shows the syntax for declaring decimal host variables.



Notes:

- 1 *level-1* indicates a COBOL level between 2 and 48.
- 2 The *picture-string* that is associated with SIGN LEADING SEPARATE must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9 or S9...9V with *i* instances of 9).
- 3 PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. The *picture-string* that is that is associated with these types must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9) or S9(*i*)V.

In COBOL, you declare the SMALLINT and INTEGER data types as a number of decimal digits. DB2 uses the full size of the integers (in a way that is similar to processing with the TRUNC(BIN) compiler option) and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration. If you compile with TRUNC(OPT) or TRUNC(STD), ensure that the size of numbers in your application is within the declared number of digits.

For small integers that can exceed 9999, use S9(4) COMP-5 or compile with TRUNC(BIN). For large integers that can exceed 999 999 999, use S9(10) COMP-3 to obtain the decimal data type. If you use COBOL for integers that exceed the COBOL PICTURE, specify the column as decimal to ensure that the data types match and perform well.

If you are using a COBOL compiler that does not support decimal numbers of more than 18 digits, use one of the following data types to hold values of greater than 18 digits:

- A decimal variable with a precision less than or equal to 18, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, the fractional part of the value might be truncated.
- An integer or a floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number might exceed the maximum value for an integer or if you want to preserve a fractional value, use a floating-point variable. Floating-point numbers

are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.

- A character-string host variable. Use the CHAR function to retrieve a decimal value into it.

Restriction: The SQL data type DECFLOAT has no equivalent in COBOL.

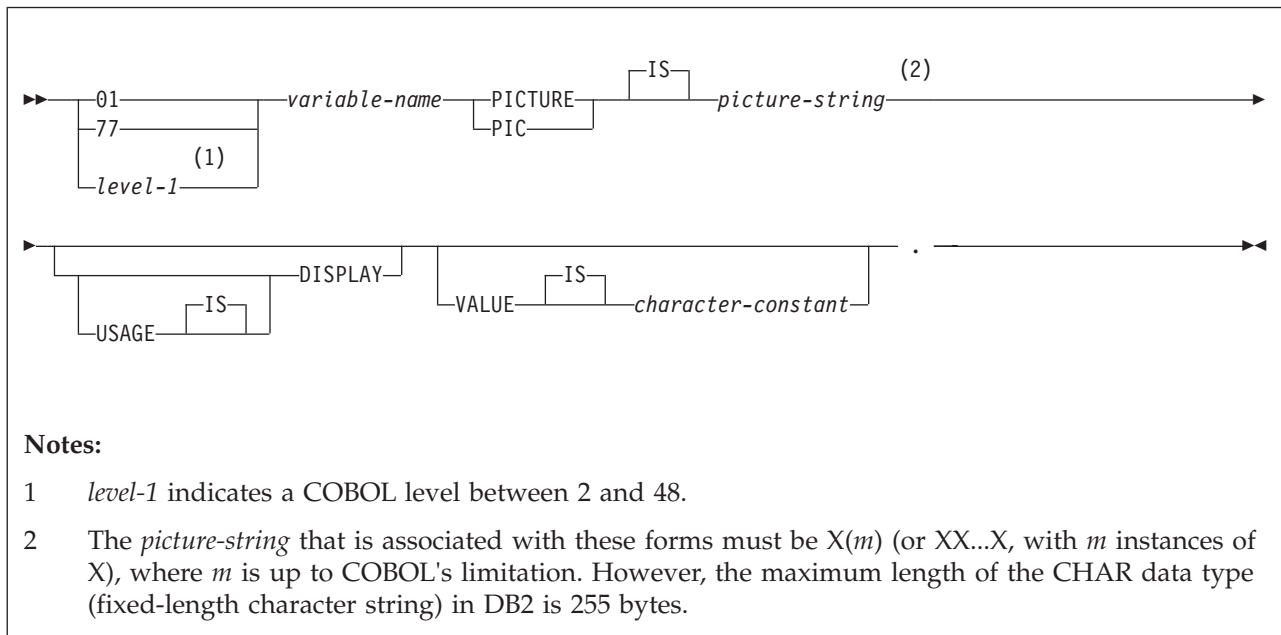
Character host variables

You can specify the following forms of character host variables:

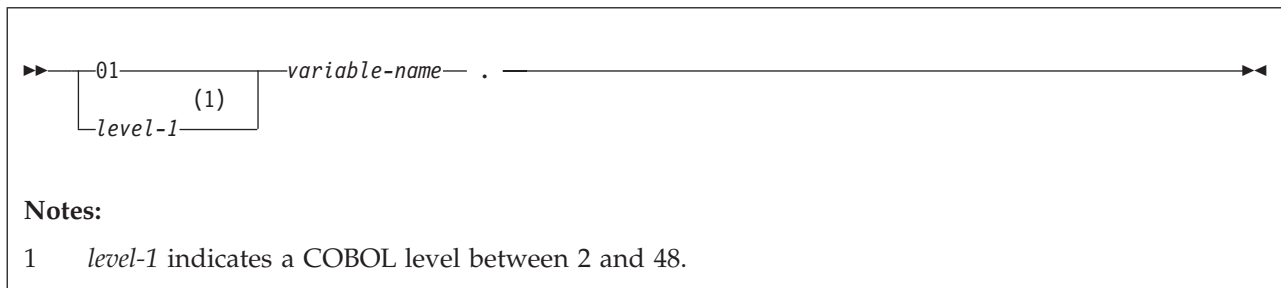
- Fixed-length strings
- Varying-length strings
- CLOBs

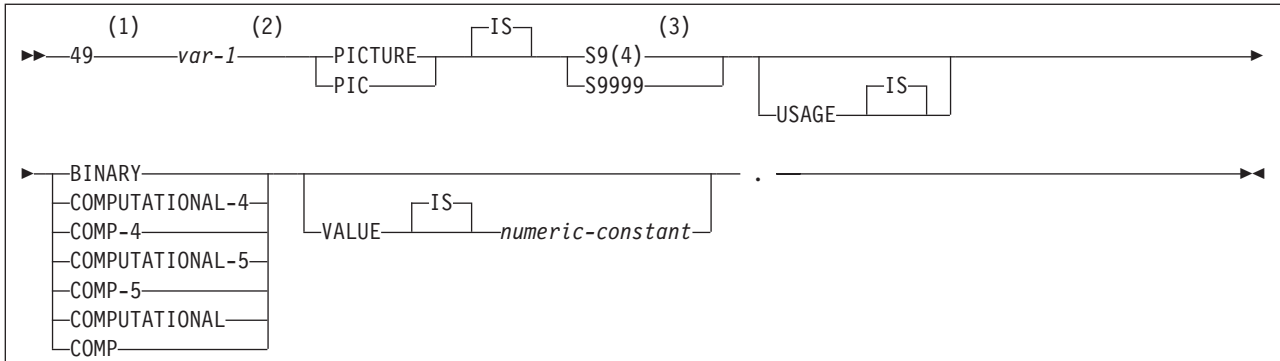
The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring fixed-length character host variables.



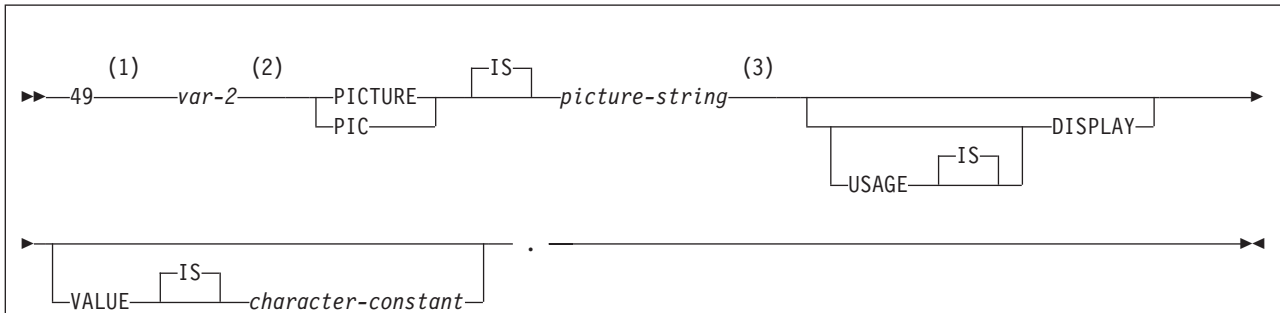
The following diagrams show the syntax for declaring varying-length character host variables.





Notes:

- 1 You cannot use an intervening REDEFINE at level 49.
- 2 You cannot directly reference *var-1* as a host variable.
- 3 DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.



Notes:

- 1 You cannot use an intervening REDEFINE at level 49.
- 2 You cannot directly reference *var-2* as a host variable.
- 3 For fixed-length strings, the *picture-string* must be *X(m)* (or *XX*, with *m* instances of *X*), where *m* is up to COBOL's limitation. However, the maximum length of the VARCHAR data type in DB2 varies depending on the data page size.

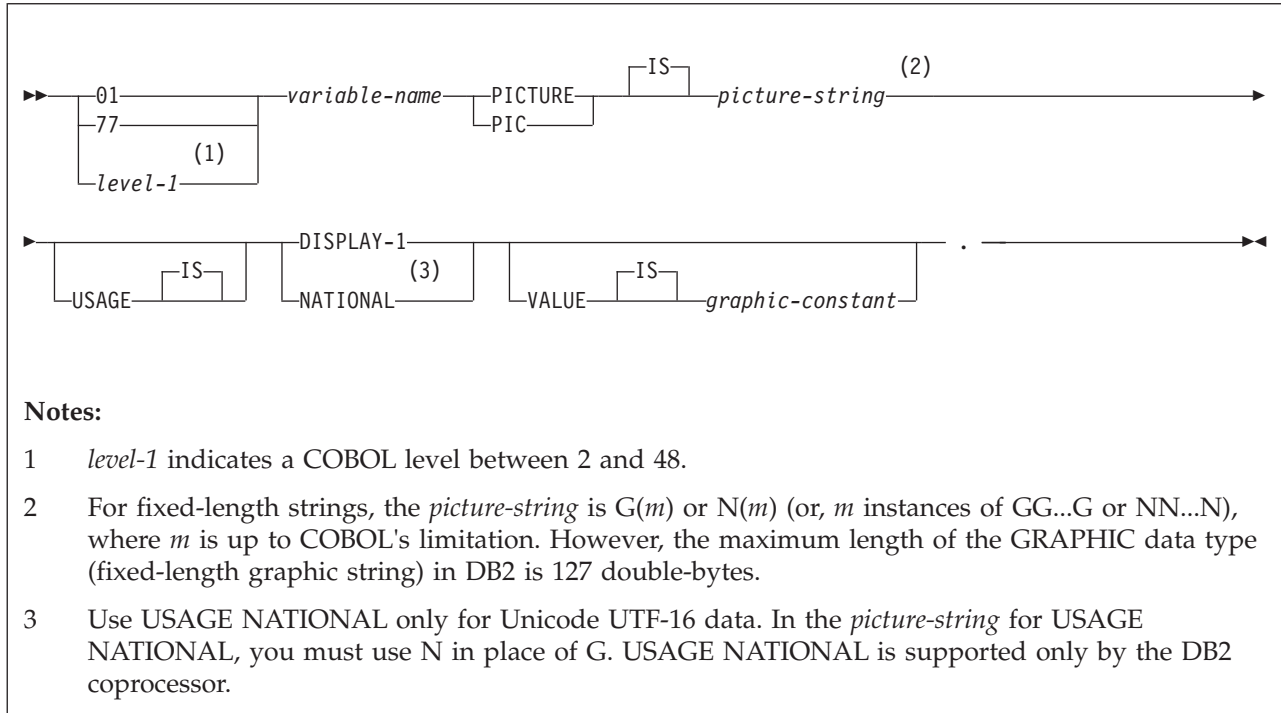
Graphic character host variables

You can specify the following forms of graphic host variables:

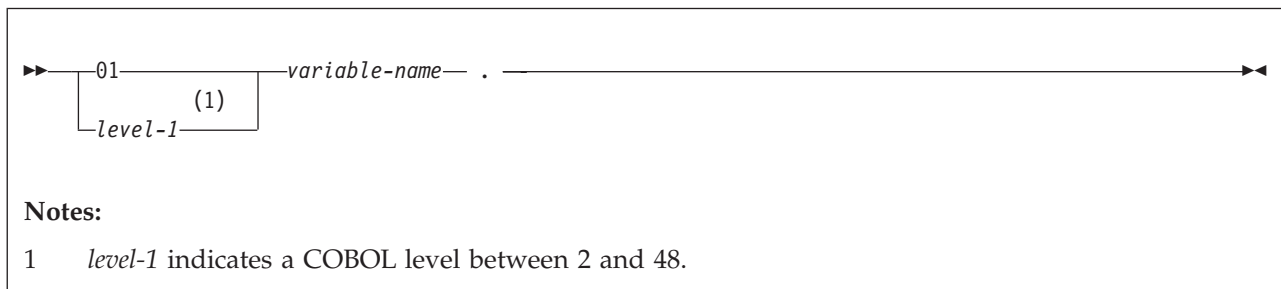
- Fixed-length strings
- Varying-length strings
- DBCLOBs

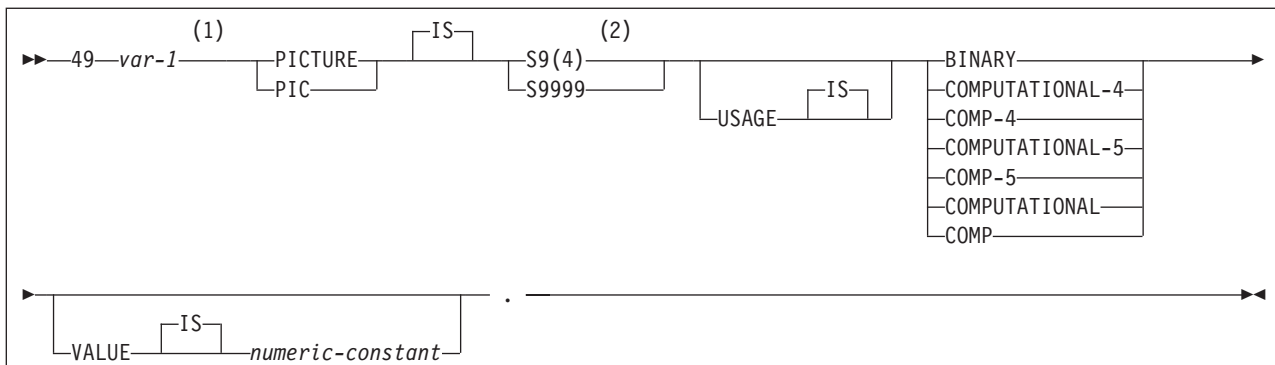
The following diagrams show the syntax for forms other than DBCLOBs.

The following diagram shows the syntax for declaring fixed-length graphic host variables.



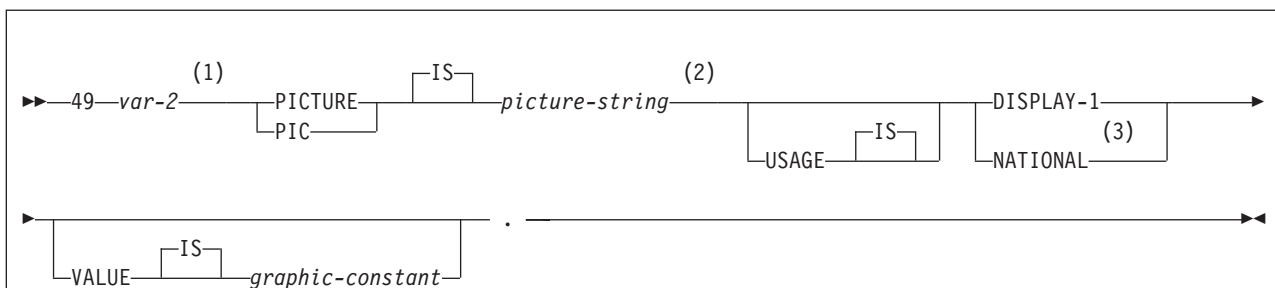
The following diagrams show the syntax for declaring varying-length graphic host variables.





Notes:

- 1 You cannot directly reference *var-1* as a host variable.
- 2 DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.



Notes:

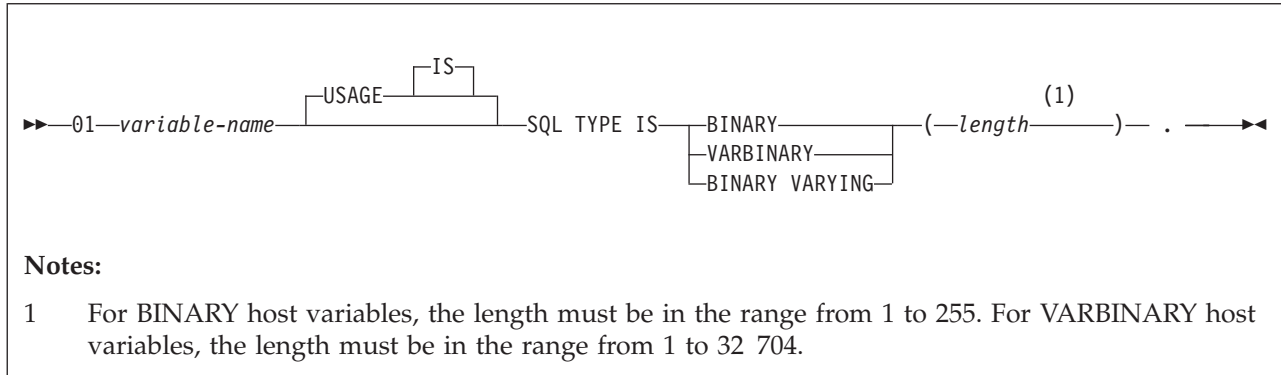
- 1 You cannot directly reference *var-2* as a host variable.
- 2 For fixed-length strings, the *picture-string* is G(*m*) or N(*m*) (or, *m* instances of GG...G or NN...N), where *m* is up to COBOL's limitation. However, the maximum length of the VARGRAPHIC data type in DB2 varies depending on the data page size.
- 3 Use USAGE NATIONAL only for Unicode UTF-16 data. In the *picture-string* for USAGE NATIONAL, you must use N in place of G. USAGE NATIONAL is supported only by the DB2 coprocessor.

Binary host variables

You can specify the following forms of binary host variables:

- Fixed-length strings
- Varying-length strings
- BLOBs

The following diagram shows the syntax for declaring BINARY and VARBINARY host variables.



COBOL does not have variables that correspond to the SQL binary types BINARY and VARBINARY. 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 COBOL language structure in the output source member.

When you reference a BINARY or VARBINARY host variable in an SQL statement, you must use the variable that you specify in the SQL TYPE declaration. When you reference the host variable in a host language statement, you must use the variable that DB2 generates.

Examples of binary variable declarations: The following table shows examples of variables that DB2 generates when you declare binary host variables.

Table 64. Examples of BINARY and VARBINARY variable declarations for COBOL

Variable declaration that you include in your COBOL program	Corresponding variable that DB2 generates in the output source member
01 BIN-VAR USAGE IS SQL TYPE IS BINARY(10).	01 BIN-VAR PIC X(10).
01 VBIN-VAR USAGE IS SQL TYPE IS VARBINARY(10).	01 VBIN-VAR. 49 VBIN-VAR-LEN PIC S9(4) USAGE BINARY. 49 VBIN-VAR-TEXT PIC X(10).

Result set locators

The following diagram shows the syntax for declaring result set locators.

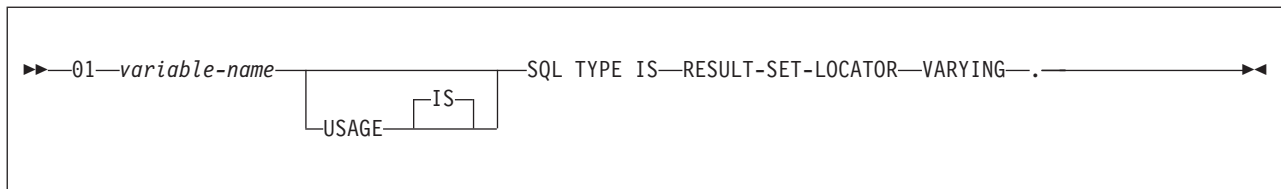
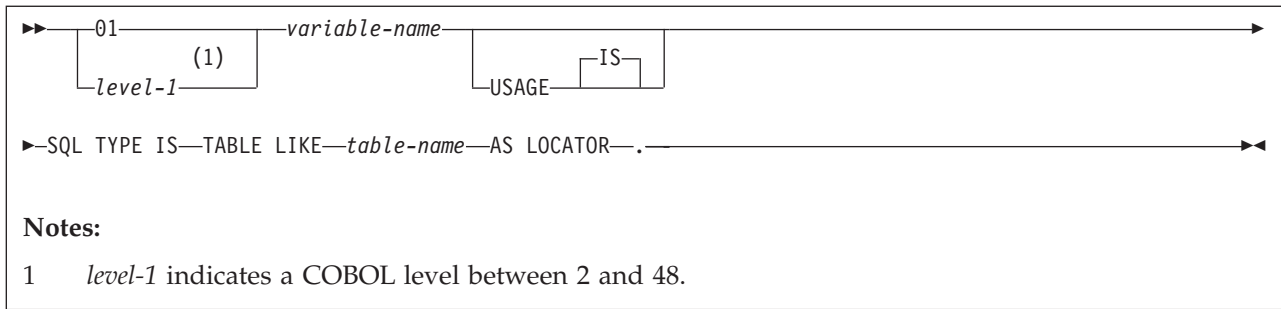


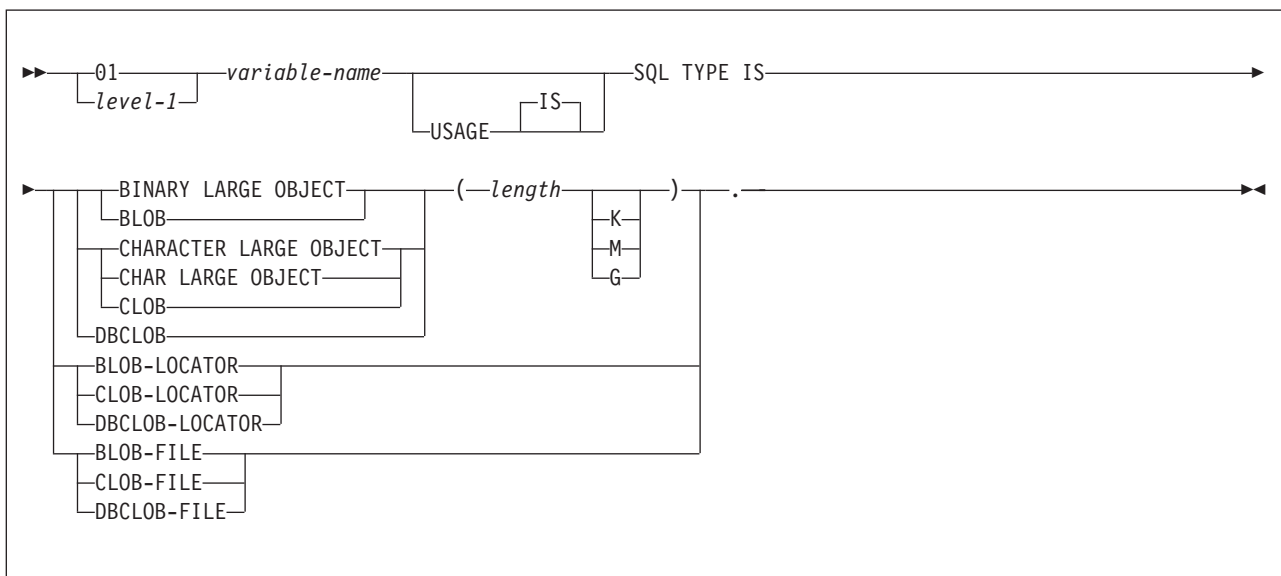
Table Locators

The following diagram shows the syntax for declaring table locators.



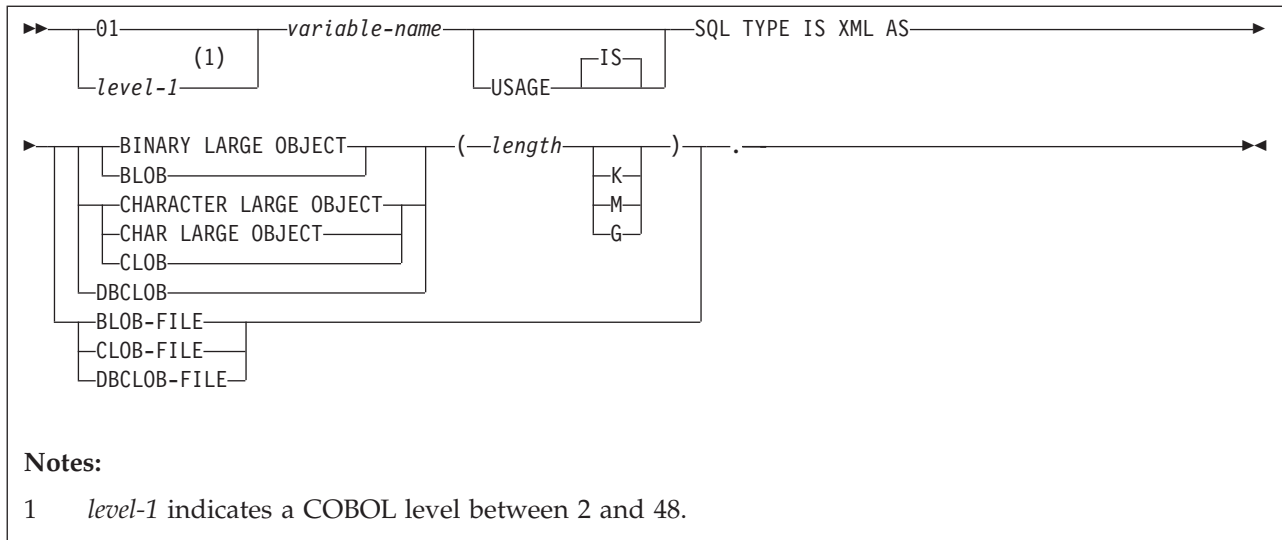
LOB variables and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB variables and file reference variables.



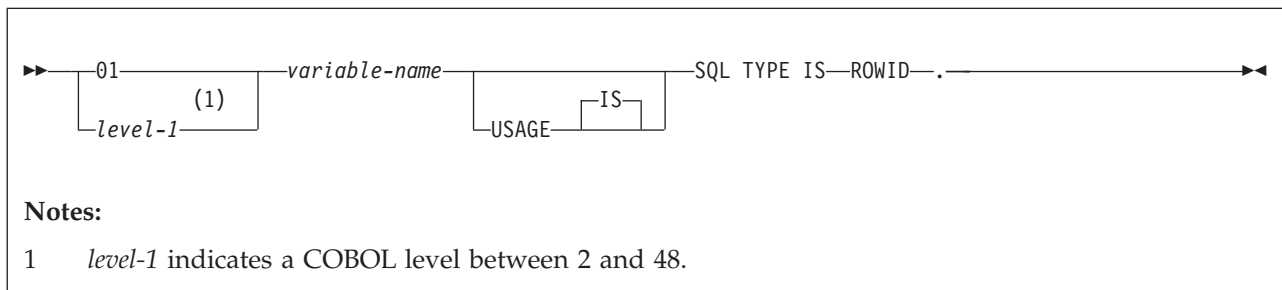
XML data host and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.



ROWID host variables

The following diagram shows the syntax for declaring ROWID host variables.



Related concepts:

“Host variables” on page 174

“Large objects (LOBs)” on page 465

Related tasks:

“Embedding SQL statements in your application” on page 183

Related reference:

 Limits in DB2 for z/OS (DB2 SQL)

Host variable arrays in COBOL

In COBOL programs, you can specify numeric, character, graphic, LOB, XML, and ROWID host variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

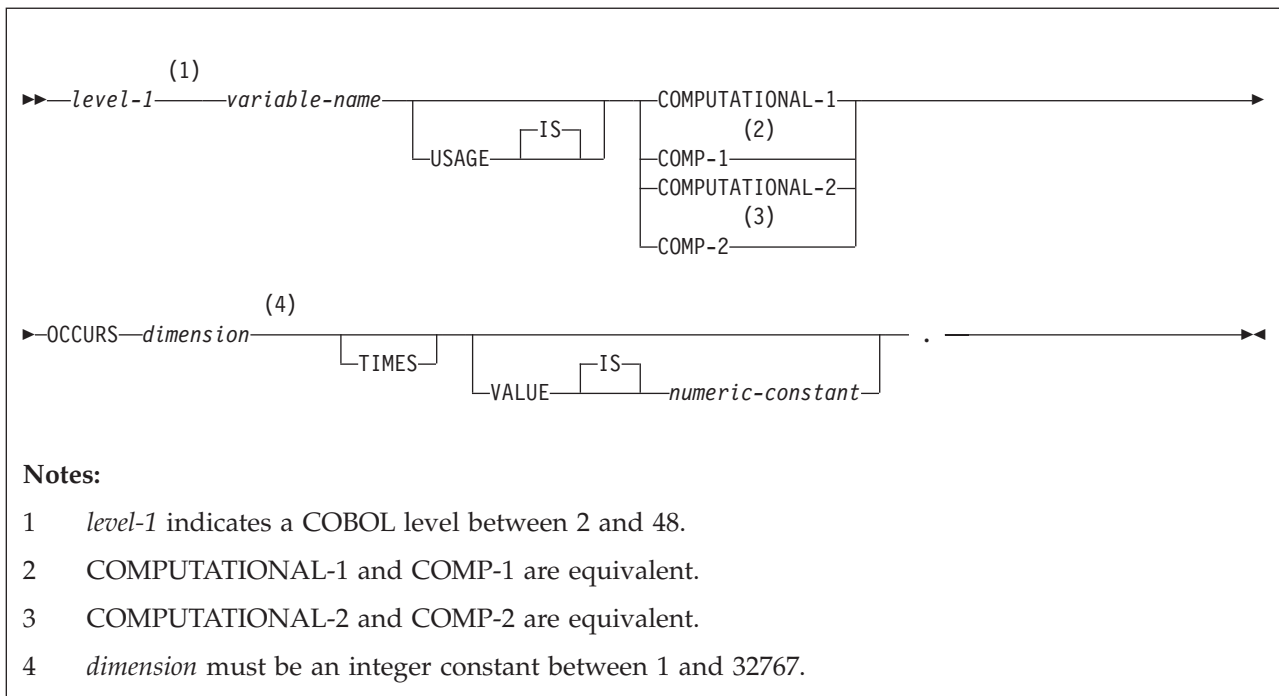
Restriction: Only some of the valid COBOL declarations are valid host variable array declarations. If the declaration for a variable array is not valid, any SQL statement that references the variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.

Numeric host variable arrays

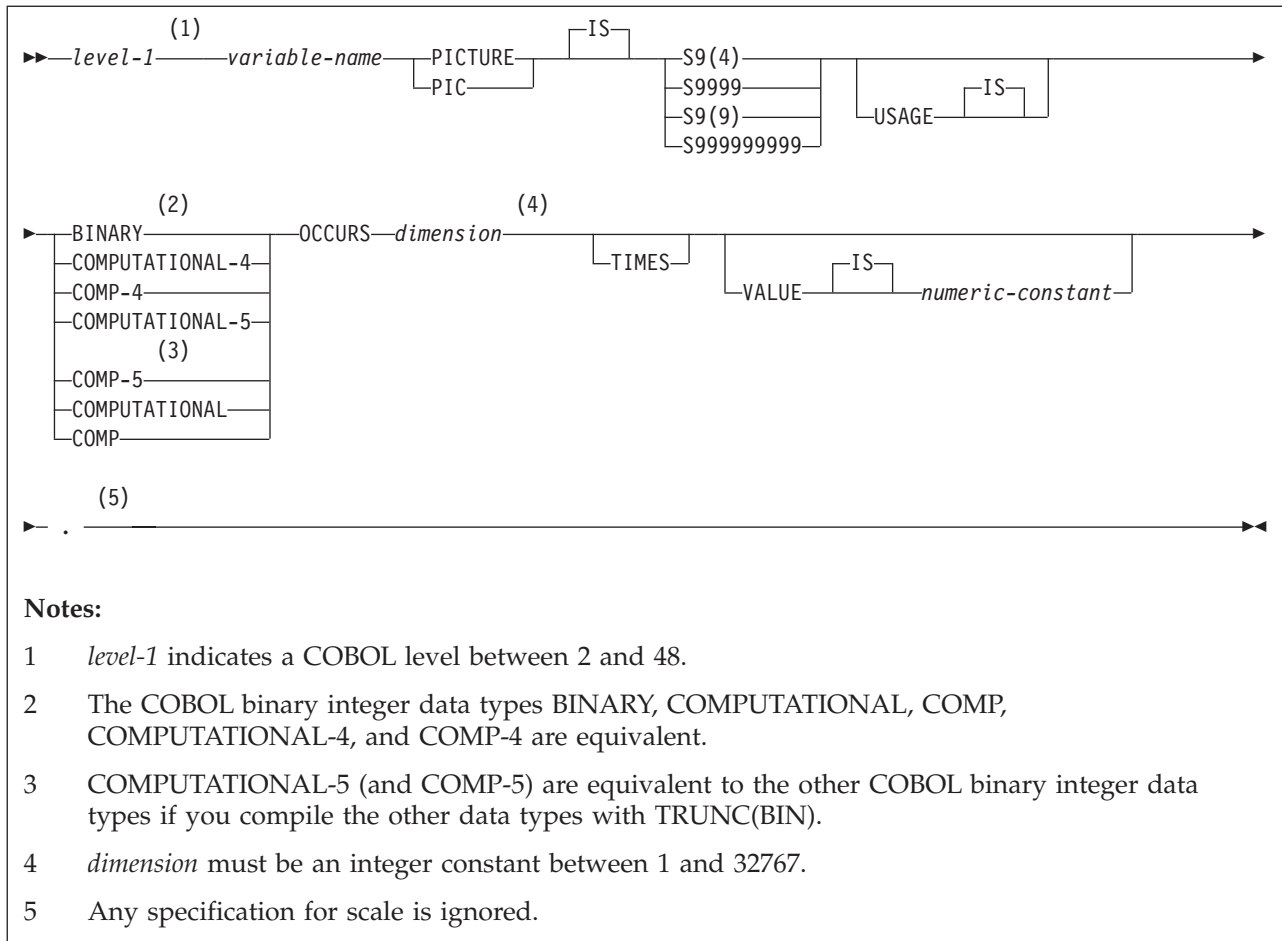
You can specify the following forms of numeric host variable arrays:

- Floating-point numbers
- Integers and small integers
- Decimal numbers

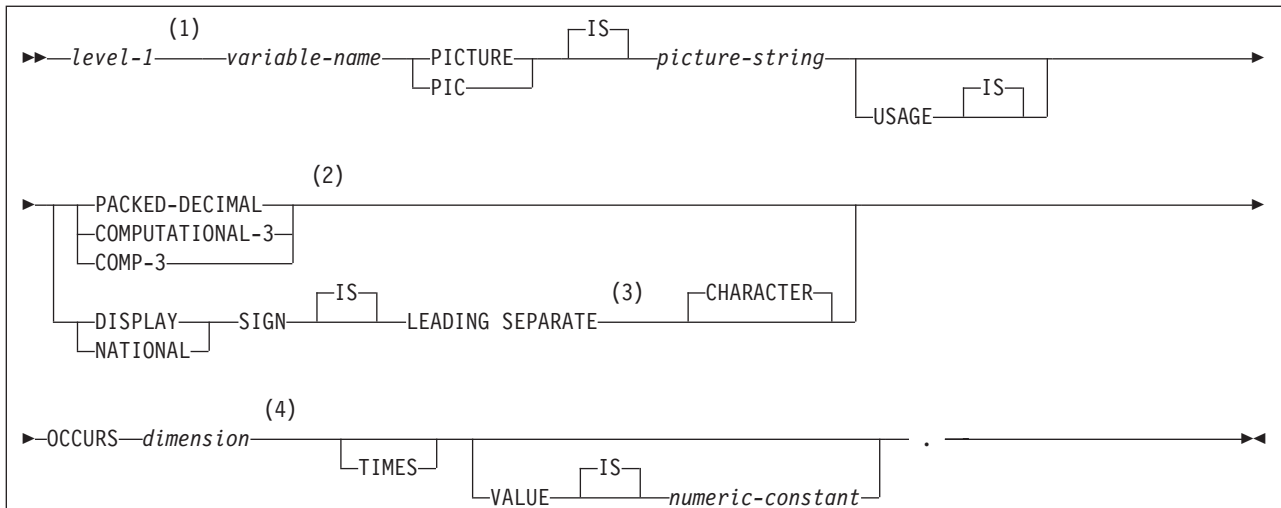
The following diagram shows the syntax for declaring floating-point host variable arrays.



The following diagram shows the syntax for declaring integer and small integer host variable arrays.



The following diagram shows the syntax for declaring decimal host variable arrays.



Notes:

- 1 *level-1* indicates a COBOL level between 2 and 48.
- 2 PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. The *picture-string* that is associated with these types must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9) or S9(*i*)V.
- 3 The *picture-string* that is associated with SIGN LEADING SEPARATE must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9 or S9...9V with *i* instances of 9).
- 4 *dimension* must be an integer constant between 1 and 32767.

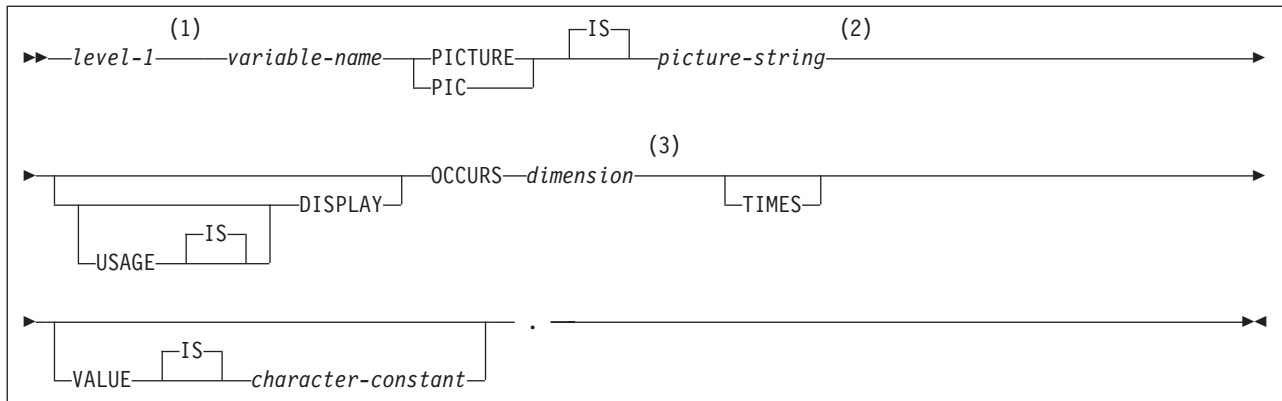
Character host variable arrays

You can specify the following forms of character host variable arrays:

- Fixed-length character strings
- Varying-length character strings
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

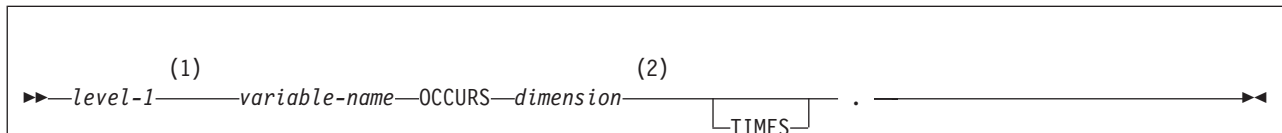
The following diagram shows the syntax for declaring fixed-length character string arrays.



Notes:

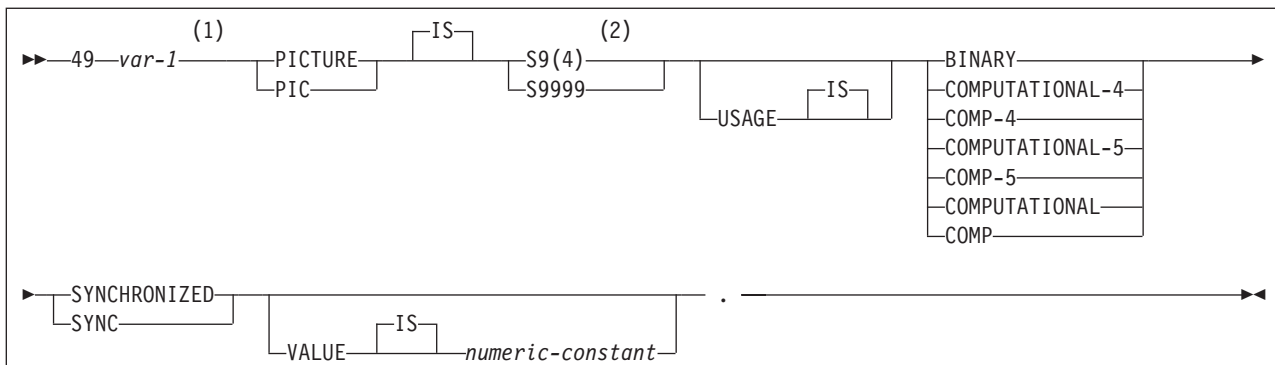
- 1 *level-1* indicates a COBOL level between 2 and 48.
- 2 The *picture-string* must be in the form *X(m)* (or *XX...X*, with *m* instances of *X*), where $1 \leq m \leq 32767$ for fixed-length strings. However, the maximum length of the CHAR data type (fixed-length character string) in DB2 is 255 bytes.
- 3 *dimension* must be an integer constant between 1 and 32767.

The following diagrams show the syntax for declaring varying-length character string arrays.



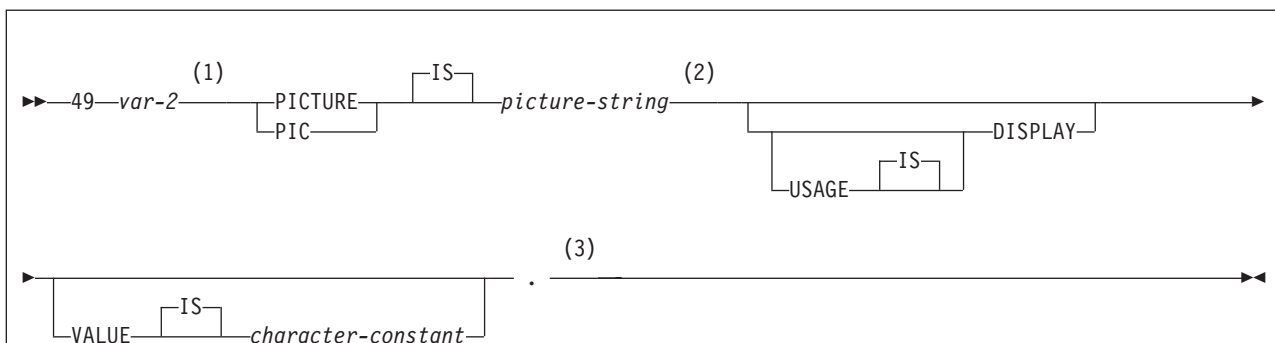
Notes:

- 1 *level-1* indicates a COBOL level between 2 and 48.
- 2 *dimension* must be an integer constant between 1 and 32767.



Notes:

- 1 You cannot directly reference *var-1* as a host variable array.
- 2 DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.



Notes:

- 1 You cannot directly reference *var-2* as a host variable array.
- 2 The *picture-string* must be in the form *X(m)* (or *XX...X*, with *m* instances of *X*), where $1 \leq m \leq 32767$ for fixed-length strings; for other strings, *m* cannot be greater than the maximum size of a varying-length character string.
- 3 You cannot use an intervening REDEFINE at level 49.

Example: The following example shows declarations of a fixed-length character array and a varying-length character array.

```

01 OUTPUT-VARS.
   05 NAME OCCURS 10 TIMES.
       49 NAME-LEN PIC S9(4) COMP-4 SYNC.
       49 NAME-DATA PIC X(40).
   05 SERIAL-NUMBER PIC S9(9) COMP-4 OCCURS 10 TIMES.

```

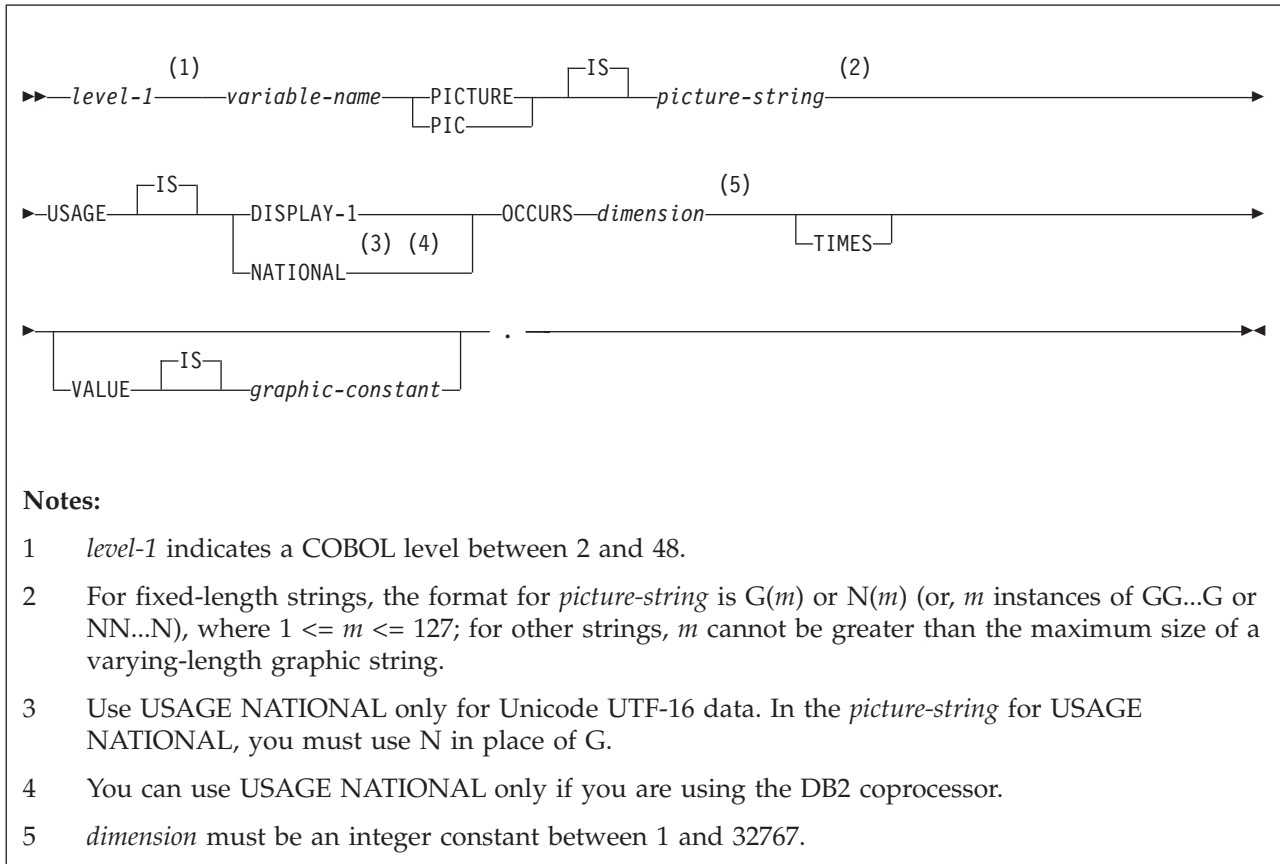
Graphic character host variable arrays

You can specify the following forms of graphic host variable arrays:

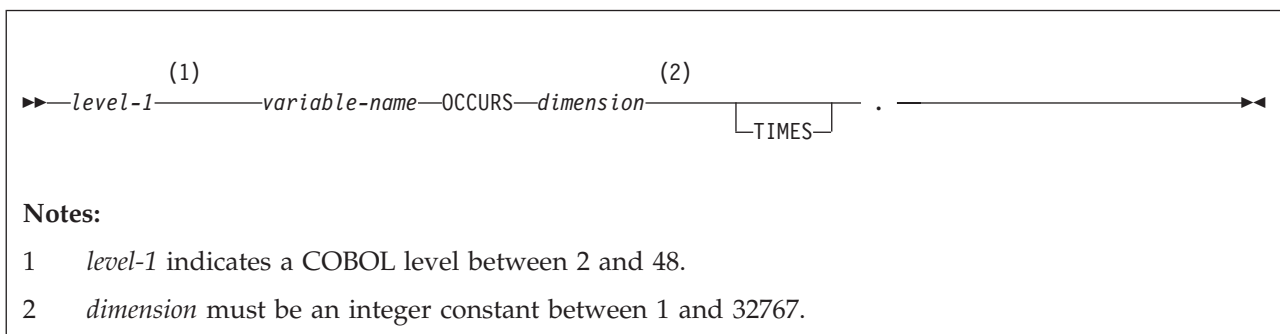
- Fixed-length strings
- Varying-length strings
- DBCLOBs

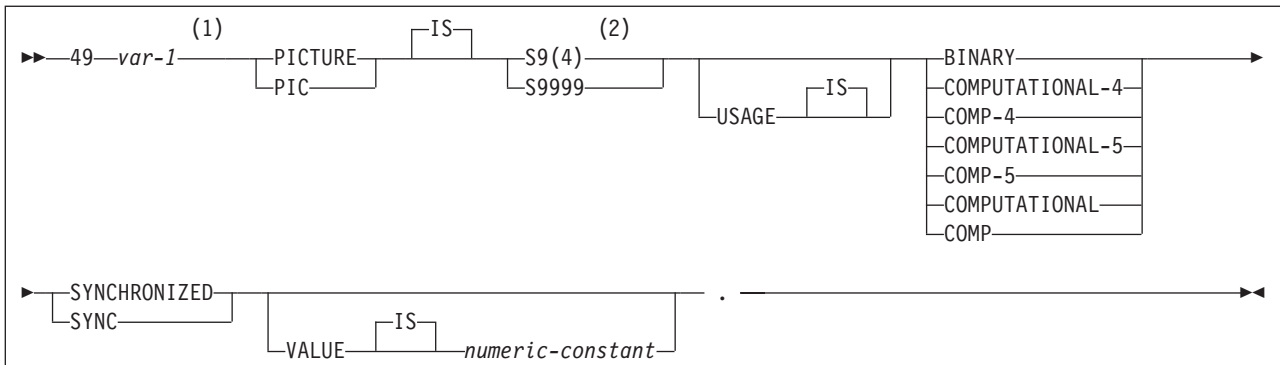
The following diagrams show the syntax for forms other than DBCLOBs.

The following diagram shows the syntax for declaring fixed-length graphic string arrays.



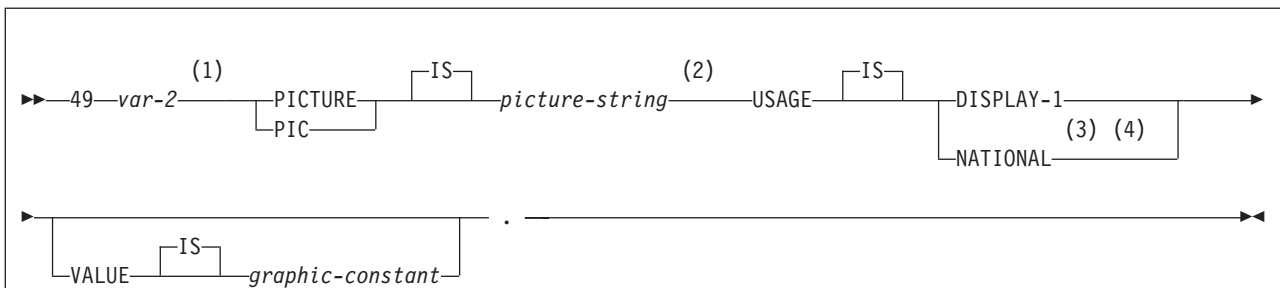
The following diagrams show the syntax for declaring varying-length graphic string arrays.





Notes:

- 1 You cannot directly reference *var-1* as a host variable array.
- 2 DB2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.

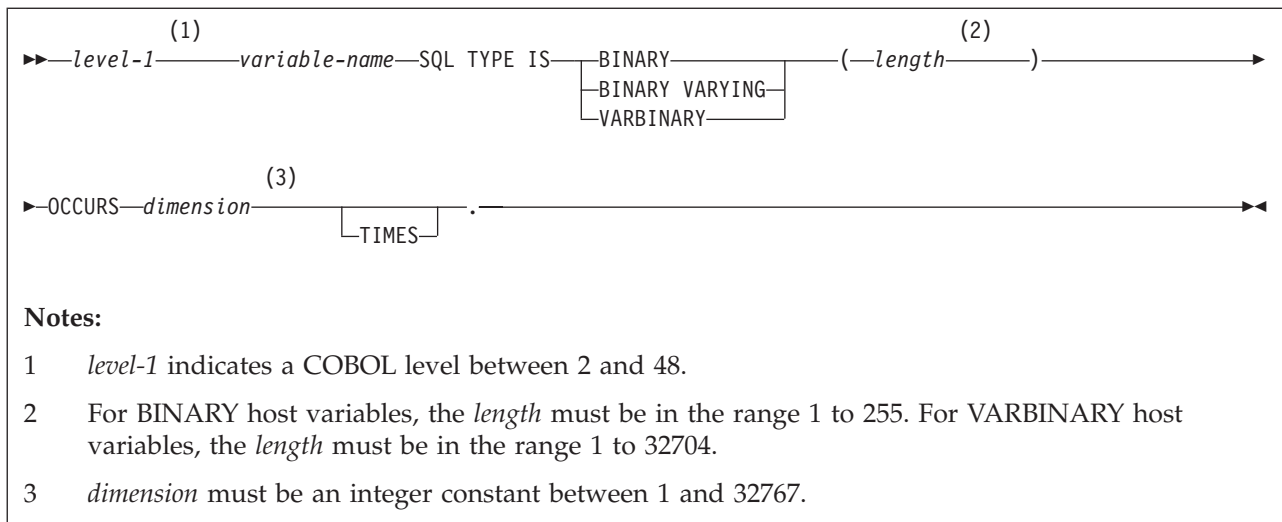


Notes:

- 1 You cannot directly reference *var-2* as a host variable array.
- 2 For fixed-length strings, the format for *picture-string* is G(*m*) or N(*m*) (or, *m* instances of GG...G or NN...N), where 1 <= *m* <= 127; for other strings, *m* cannot be greater than the maximum size of a varying-length graphic string.
- 3 Use USAGE NATIONAL only for Unicode UTF-16 data. In the *picture-string* for USAGE NATIONAL, you must use N in place of G.
- 4 You can use USAGE NATIONAL only if you are using the DB2 coprocessor.

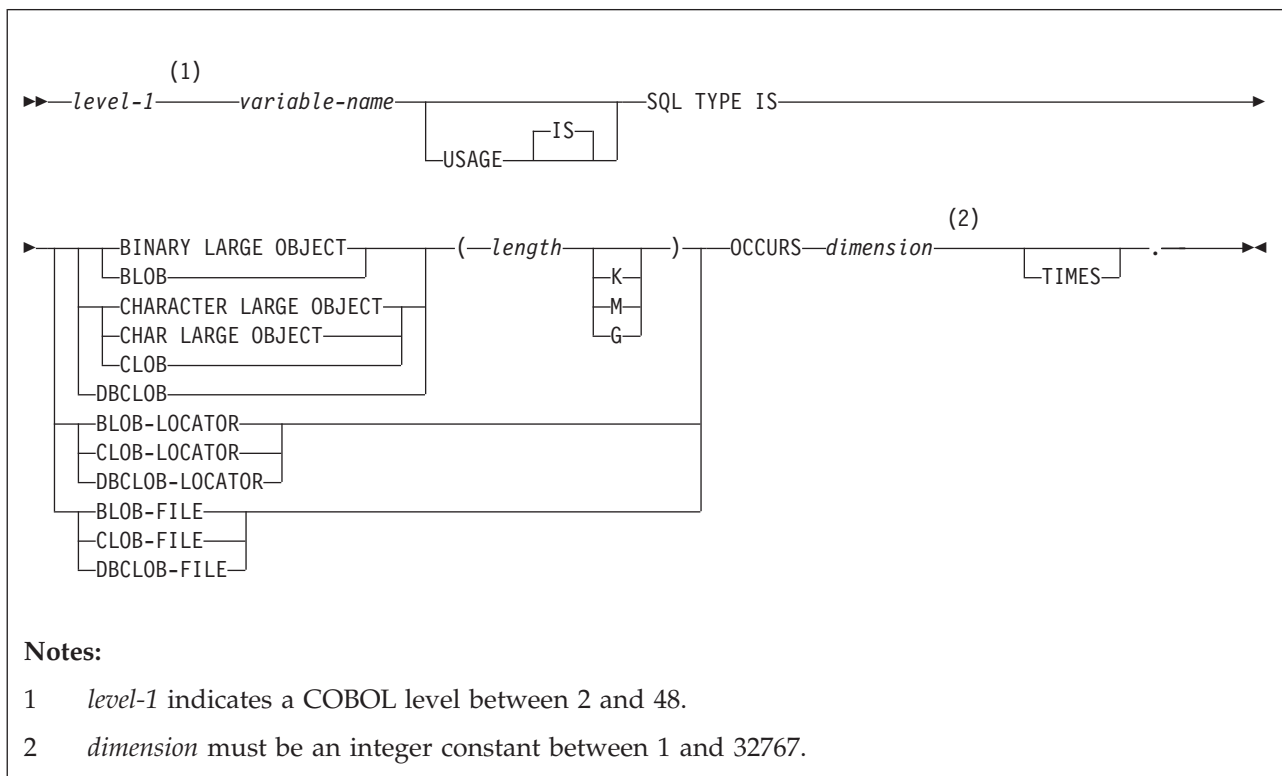
Binary host variable arrays

The following diagram shows the syntax for declaring binary host variable arrays.



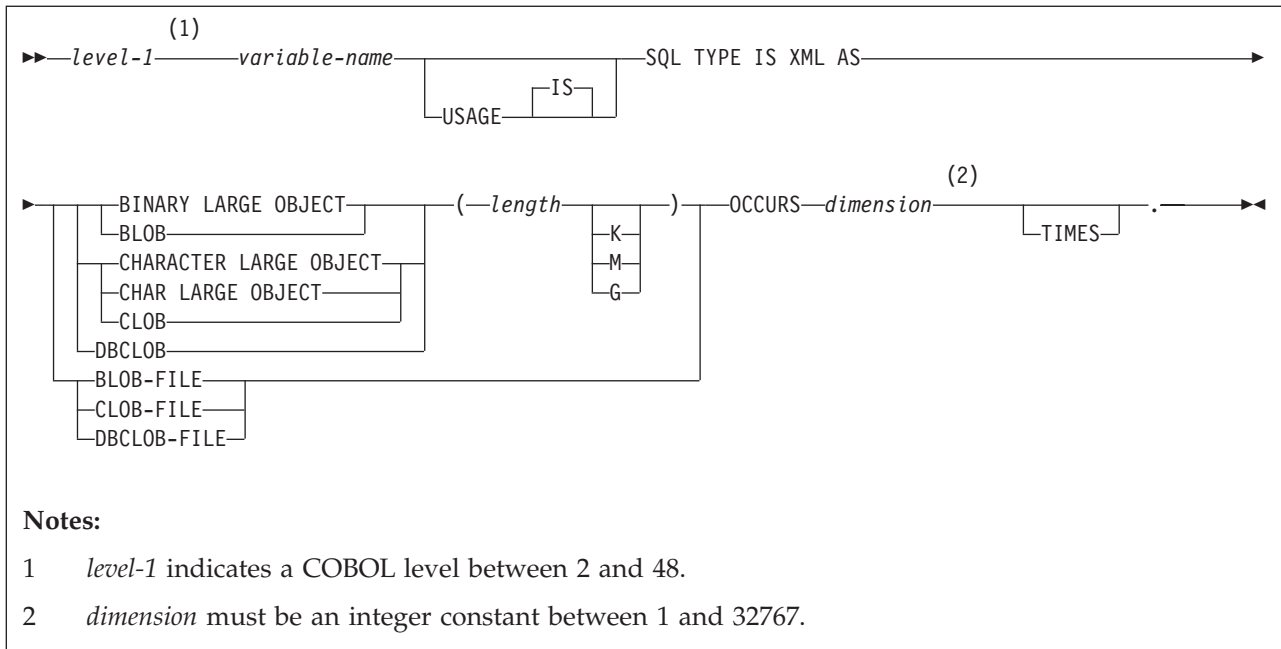
LOB, locator, and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable, locator, and file reference arrays.



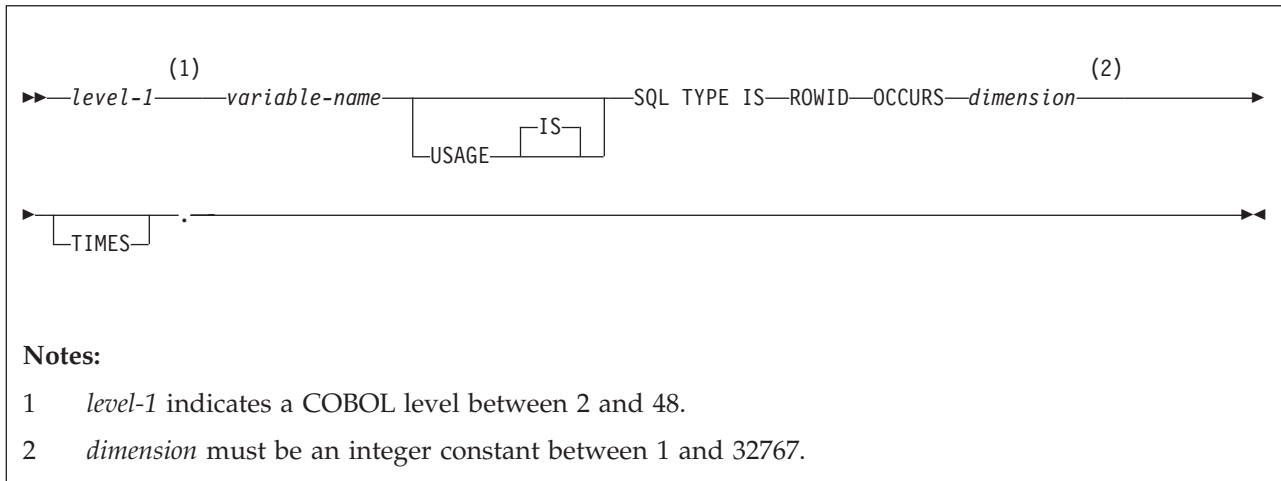
XML host and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable and file reference arrays for XML data types.



ROWID variable arrays

The following diagram shows the syntax for declaring ROWID variable arrays.



Related concepts:

“Host variable arrays in an SQL statement” on page 191

“Host variable arrays” on page 175

“Large objects (LOBs)” on page 465

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

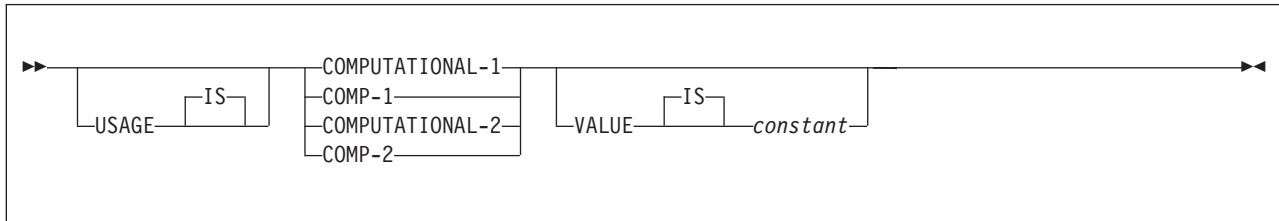
“Retrieving multiple rows of data into host variable arrays” on page 192

Host structures in COBOL

A COBOL host structure is a named set of host variables that are defined in your program's WORKING-STORAGE SECTION or LINKAGE SECTION.

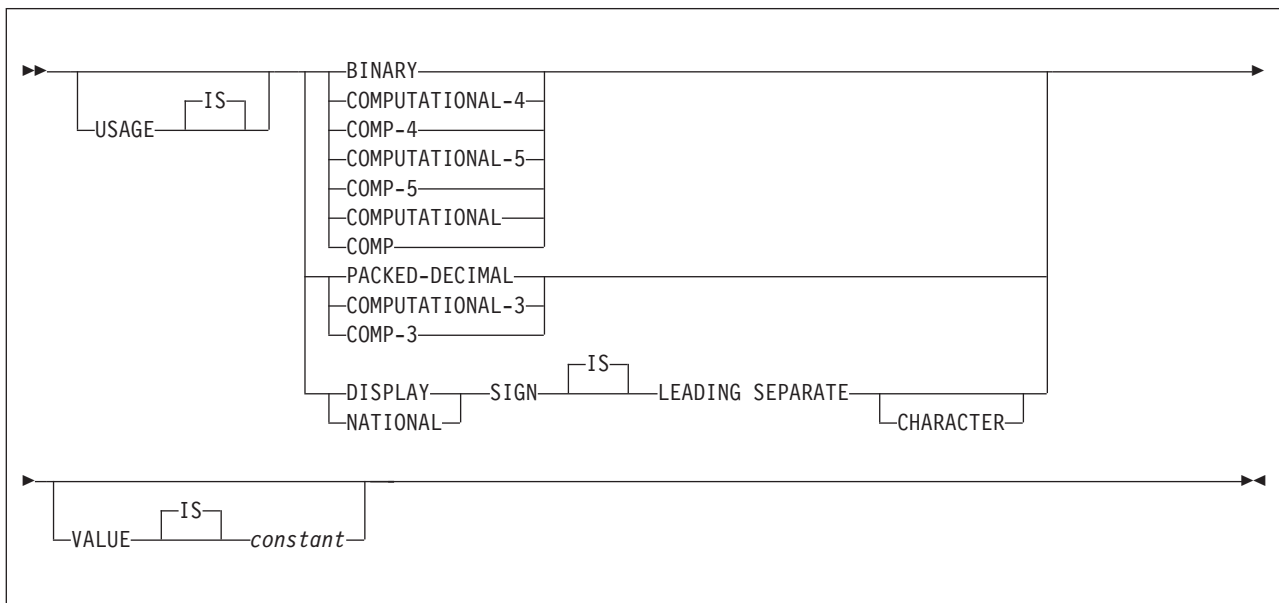
Numeric usage items

The following diagram shows the syntax for numeric-usage items that are used within declarations of host structures.



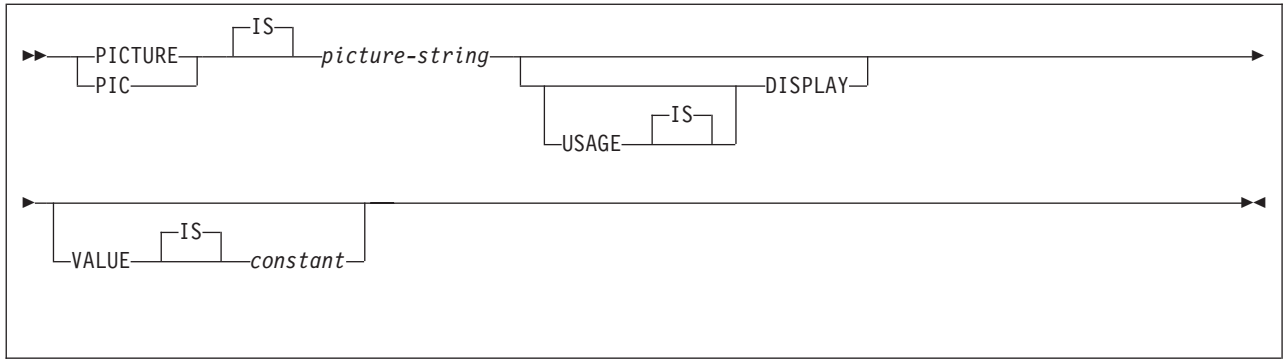
Integer and decimal usage items

The following diagram shows the syntax for integer and decimal usage items that are used within declarations of host structures.



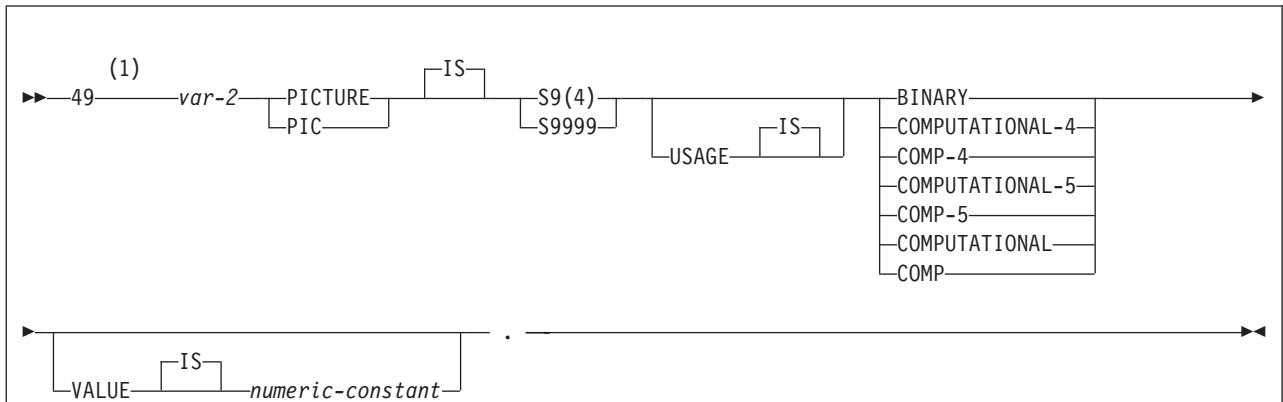
CHAR inner variables

The following diagram shows the syntax for CHAR inner variables that are used within declarations of host structures.



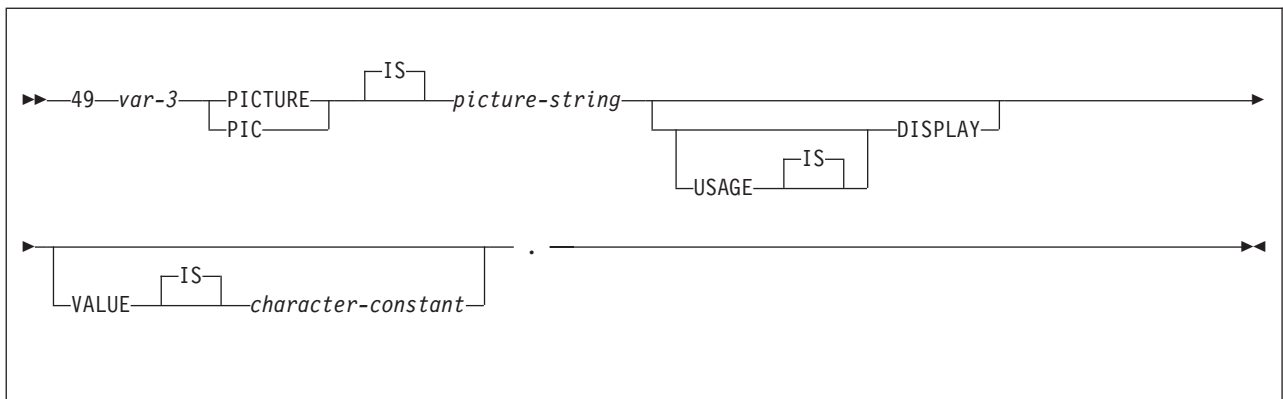
VARCHAR inner variables

The following diagrams show the syntax for VARCHAR inner variables that are used within declarations of host structures.



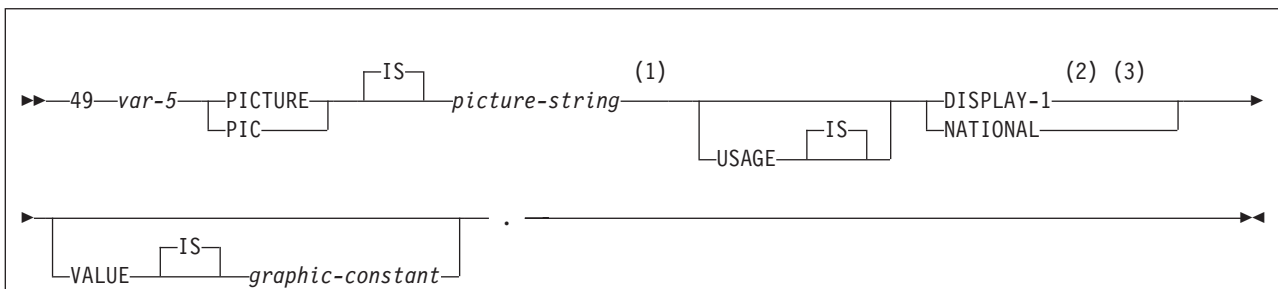
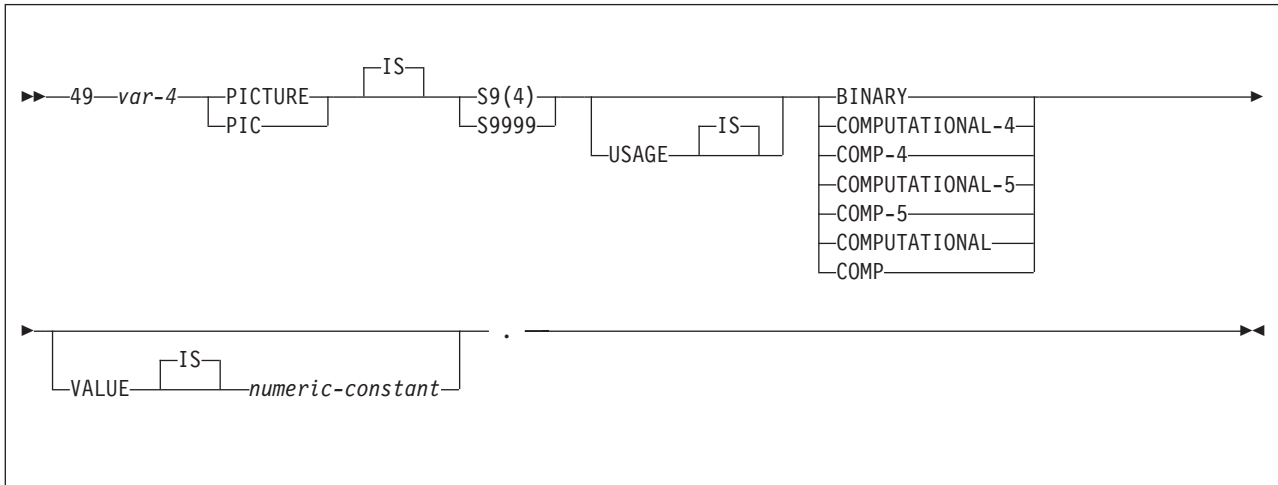
Notes:

- 1 The number 49 has a special meaning to DB2. Do not specify another number.



VARGRAPHIC inner variables

The following diagrams show the syntax for VARGRAPHIC inner variables that are used within declarations of host structures.

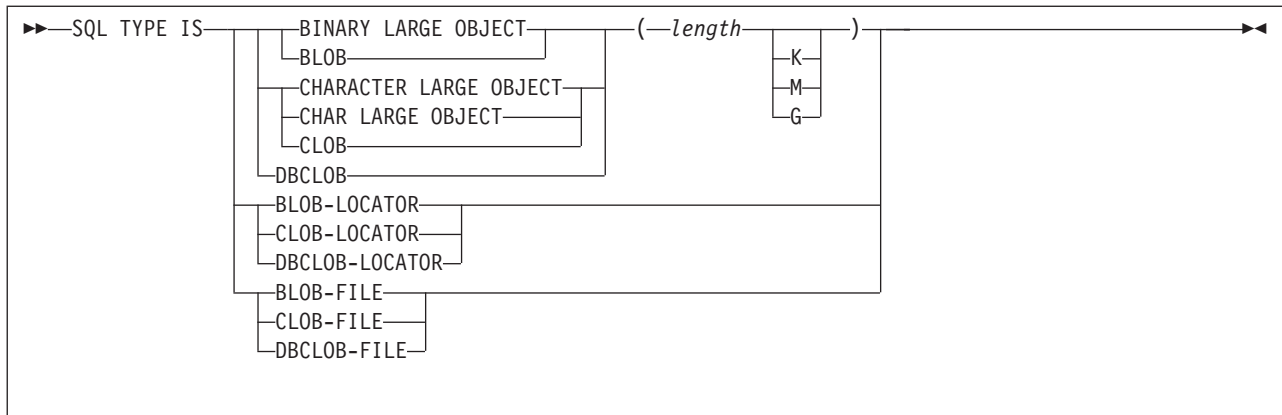


Notes:

- 1 For fixed-length strings, the format of *picture-string* is $G(m)$ or $N(m)$ (or, m instances of $GG...G$ or $NN...N$), where $1 \leq m \leq 127$; for other strings, m cannot be greater than the maximum size of a varying-length graphic string.
- 2 Use `USAGE NATIONAL` for only Unicode UTF-16 data. In the *picture-string* for `USAGE NATIONAL`, you must use `N` in place of `G`.
- 3 You can use `USAGE NATIONAL` only if you are using the DB2 coprocessor.

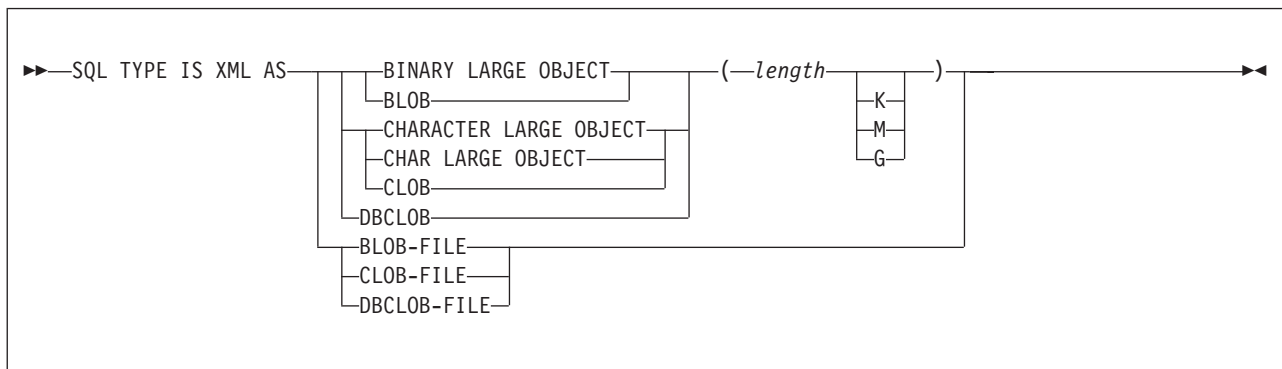
LOB variables, locators, and file reference variables

The following diagram shows the syntax for LOB variables, locators, and file reference variables that are used within declarations of host structures.



LOB variables and file reference variables for XML data

The following diagram shows the syntax for LOB variables and file reference variables that are used within declarations of host structures for XML.



Example

In the following example, B is the name of a host structure that contains the elementary items C1 and C2.

```
01 A
  02 B
    03 C1 PICTURE ...
    03 C2 PICTURE ...
```

To reference the C1 field in an SQL statement, specify B.C1.

Related concepts:

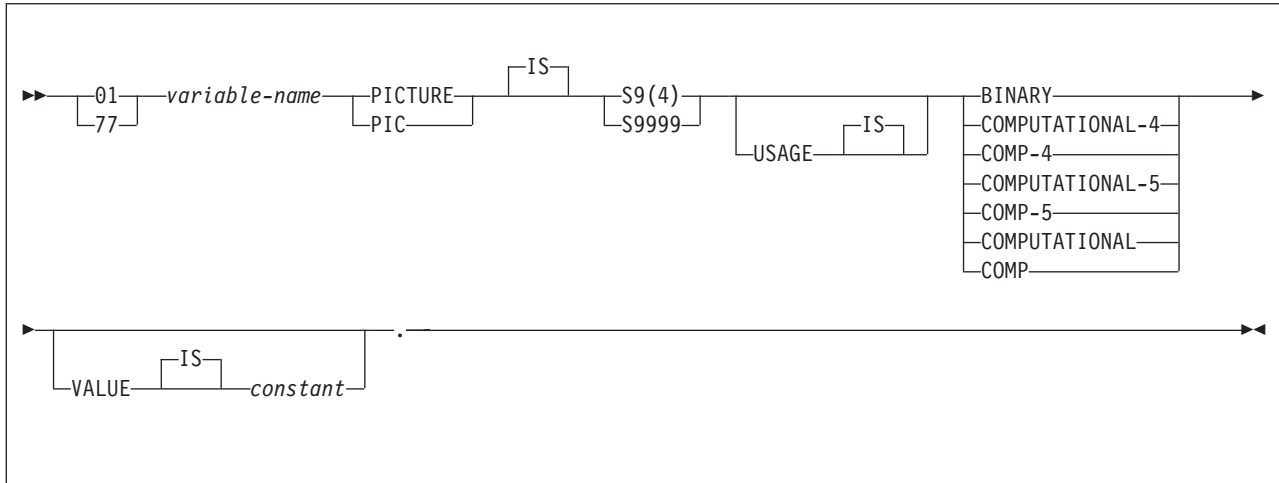
“Host structures” on page 175

Indicator variables, indicator arrays, and host structure indicator arrays in COBOL

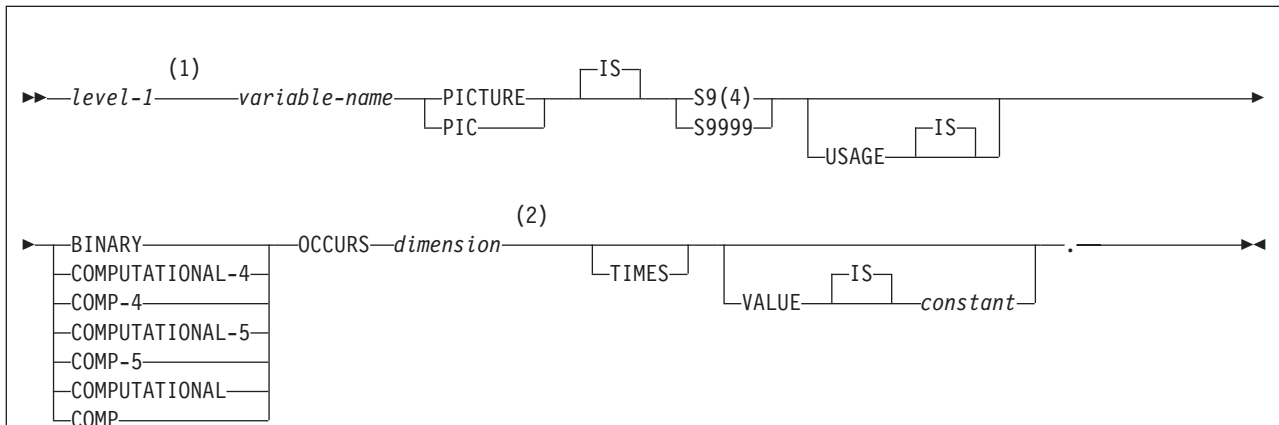
An indicator variable is a 2-byte integer (PIC S9(4) USAGE BINARY). An indicator variable array is an array of 2-byte integers (PIC S9(4) USAGE BINARY). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

You can define indicator variables as scalar variables or as array elements in a structure form or as an array variable by using a single level OCCURS clause.

The following diagram shows the syntax for declaring an indicator variable in COBOL.



The following diagram shows the syntax for declaring an indicator array in COBOL.



Notes:

- 1 *level-1* must be an integer between 2 and 48.
- 2 *dimension* must be an integer constant between 1 and 32767.

Example

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

```
EXEC SQL FETCH CLS_CURSOR INTO :CLS-CD,
                                :DAY :DAY-IND,
                                :BGN :BGN-IND,
                                :END :END-IND
END-EXEC.
```

You can declare these variables as follows:


```
77 CLS-CD    PIC X(7).
77 DAY      PIC S9(4) BINARY.
77 BGN      PIC X(8).
77 END      PIC X(8).
77 DAY-IND  PIC S9(4) BINARY.
77 BGN-IND  PIC S9(4) BINARY.
77 END-IND  PIC S9(4) BINARY.
```

Related concepts:

“Indicator variables, arrays, and structures” on page 176

Related tasks:

“Inserting null values into columns by using indicator variables or arrays” on page 190

Controlling the CCSID for COBOL host variables

Setting the CCSID for COBOL host variables is slightly different than the process for other host languages. In COBOL, several other settings affect the CCSID.

This task applies to programs that use IBM Enterprise COBOL for z/OS and the DB2 coprocessor.

To control the CCSID for COBOL host variables:

Use one or more of the following items:

The NATIONAL data type

Use this data type to declare Unicode values in the UTF-16 format (CCSID 1200).

If you declare a host variable HV1 as USAGE NATIONAL, DB2 always handles HV1 as if you had used the following DECLARE VARIABLE statement:

```
DECLARE :HV1 VARIABLE CCSID 1200
```

The COBOL CODEPAGE compiler option

Use this option to specify the default EBCDIC CCSID of character data items.

The SQLCCSID compiler option

Use this option to control whether the CODEPAGE compiler option influences the processing of SQL host variables in your COBOL programs (available in Enterprise COBOL V3R4 or later).

When you specify the SQLCCSID compiler option, the COBOL DB2 coprocessor uses the CCSID that is specified in the CODEPAGE compiler option. All host variables of character data type, other than NATIONAL, are specified with that CCSID unless they are explicitly overridden by a DECLARE VARIABLE statement.

When you specify the NOSQLCCSID compiler option, the CCSID that is specified in the CODEPAGE compiler option is used for processing only COBOL statements within the COBOL program. That CCSID is not used for the processing of host variables in SQL statements. DB2 uses the CCSIDs that are specified through DB2 mechanisms and defaults as host variable data value encodings.

The DECLARE VARIABLE statement.

This statement explicitly sets the CCSID for individual host variables.

Assume that the COBOL SQLCCSID compiler option is specified and that the COBOL CODEPAGE compiler option is specified as CODEPAGE(1141). The following code shows how you can control the CCSID:

```
DATA DIVISION.
  01 HV1 PIC N(10) USAGE NATIONAL.
  01 HV2 PIC X(20) USAGE DISPLAY.
  01 HV3 PIC X(30) USAGE DISPLAY.
  ...
  EXEC SQL
    DECLARE :HV3 VARIABLE CCSID 1047
  END-EXEC.
  ...
PROCEDURE DIVISION.
  ...
  EXEC SQL
    SELECT C1, C2, C3 INTO :HV1, :HV2, :HV3 FROM T1
  END-EXEC.
```

Each of the host variables have the following CCSIDs:

```
HV1  1200
HV2  1141
HV3  1047
```

Assume that the COBOL NOSQLCCSID compiler option is specified, the COBOL CODEPAGE compiler option is specified as CODEPAGE(1141), and the DB2 default single byte CCSID is set to 37. In this case, each of the host variables in this example have the following CCSIDs:

```
HV1  1200
HV2  37
HV3  1047
```

Related reference:

“Host variables in COBOL” on page 326

 [Compiler options \(COBOL\) \(Enterprise COBOL for z/OS Programming Guide\)](#)

Equivalent SQL and COBOL data types

When you declare host variables in your COBOL programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

Table 65. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in COBOL programs

COBOL host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
COMP-1	480	4	REAL or FLOAT(<i>n</i>) 1<= <i>n</i> <=21
COMP-2	480	8	DOUBLE PRECISION, or FLOAT(<i>n</i>) 22<= <i>n</i> <=53

Table 65. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in COBOL programs (continued)

COBOL host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
S9(i)V9(d) COMP-3 or S9(i)V9(d) PACKED-DECIMAL	484	$i+d$ in byte 1, d in byte 2	DECIMAL($i+d,d$) or NUMERIC($i+d,d$)
S9(i)V9(d) DISPLAY SIGN LEADING SEPARATE	504	$i+d$ in byte 1, d in byte 2	No exact equivalent. Use DECIMAL($i+d,d$) or NUMERIC($i+d,d$)
S9(i)V9(d) NATIONAL SIGN LEADING SEPARATE	504	$i+d$ in byte 1, d in byte 2	No exact equivalent. Use DECIMAL($i+d,d$) or NUMERIC($i+d,d$)
S9(4) COMP-4, S9(4) COMP-5, S9(4) COMP, or S9(4) BINARY	500	2	SMALLINT
S9(9) COMP-4, S9(9) COMP-5, S9(9) COMP, or S9(9) BINARY	496	4	INTEGER
S9(18) COMP-4, S9(18) COMP-5, S9(18) COMP, or S9(18) BINARY	492	8	BIGINT
Fixed-length character data	452	n	CHAR(n)
Varying-length character data $1 \leq n \leq 255$	448	n	VARCHAR(n)
Varying-length character data $m > 255$	456	m	VARCHAR(m)
Fixed-length graphic data	468	m	GRAPHIC(m)
Varying-length graphic data $1 \leq m \leq 127$	464	m	VARGRAPHIC(m)
Varying-length graphic data $m > 127$	472	m	VARGRAPHIC(m)
SQL TYPE is BINARY(n), $1 \leq n \leq 255$	912	n	BINARY(n)
SQL TYPE is VARBINARY(n), $1 \leq n \leq 32\ 704$	908	n	VARBINARY(n)
SQL TYPE IS RESULT-SET-LOCATOR	972	4	Result set locator ²
SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	976	4	Table locator ²
SQL TYPE IS BLOB-LOCATOR	960	4	BLOB locator ²
SQL TYPE IS CLOB-LOCATOR	964	4	CLOB locator ²
SQL TYPE IS DBCLOB-LOCATOR	968	4	DBCLOB locator ²
USAGE IS SQL TYPE IS BLOB(n) $1 \leq n \leq 2147483647$	404	n	BLOB(n)
USAGE IS SQL TYPE IS CLOB(n) $1 \leq n \leq 2147483647$	408	n	CLOB(n)
USAGE IS SQL TYPE IS DBCLOB(m) $1 \leq m \leq 1073741823^3$	412	n	DBCLOB(m) ³
SQL TYPE IS XML AS BLOB(n)	404	0	XML
SQL TYPE IS XML AS CLOB(n)	408	0	XML

Table 65. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in COBOL programs (continued)

COBOL host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
SQL TYPE IS XML AS DBCLOB(<i>n</i>)	412	0	XML
SQL TYPE IS BLOB-FILE	916/917	267	BLOB file reference ²
SQL TYPE IS CLOB-FILE	920/921	267	CLOB file reference ²
SQL TYPE IS DBCLOB-FILE	924/925	267	DBCLOB file reference ²
SQL TYPE IS XML AS BLOB-FILE	916/917	267	XML BLOB file reference ²
SQL TYPE IS XML AS CLOB-FILE	920/921	267	XML CLOB file reference ²
SQL TYPE IS XML AS DBCLOB-FILE	924/925	267	XML DBCLOB file reference ²
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
2. Do not use this data type as a column type.
3. *m* is the number of double-byte characters.

The following table shows equivalent COBOL host variables for each SQL data type. Use this table to determine the COBOL data type for host variables that you define to receive output from the database. For example, if you retrieve **TIMESTAMP** data, you can define a fixed-length character string variable of length *n*

This table shows direct conversions between SQL data types and COBOL data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 converts those compatible data types.

Table 66. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type

SQL data type	COBOL host variable equivalent	Notes
SMALLINT	S9(4) COMP-4, S9(4) COMP-5, S9(4) COMP, or S9(4) BINARY	
INTEGER	S9(9) COMP-4, S9(9) COMP-5, S9(9) COMP, or S9(9) BINARY	
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	S9(<i>p-s</i>)V9(<i>s</i>) COMP-3 or S9(<i>p-s</i>)V9(<i>s</i>) PACKED-DECIMAL DISPLAY SIGN LEADING SEPARATE NATIONAL SIGN LEADING SEPARATE	<i>p</i> is precision; <i>s</i> is scale. 0<= <i>s</i> <= <i>p</i> <=31. If <i>s</i> =0, use S9(<i>p</i>)V or S9(<i>p</i>). If <i>s</i> = <i>p</i> , use SV9(<i>s</i>). If the COBOL compiler does not support 31-digit decimal numbers, no exact equivalent exists. Use COMP-2.
REAL or FLOAT (<i>n</i>)	COMP-1	1<= <i>n</i> <=21

Table 66. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	COBOL host variable equivalent	Notes
DOUBLE PRECISION, DOUBLE or FLOAT (<i>n</i>)	COMP-2	22<= <i>n</i> <=53
BIGINT	S9(18) COMP-4, S9(18) COMP-5, S9(18) COMP, or S9(18) BINARY	
CHAR(<i>n</i>)	Fixed-length character string. For example, 01 VAR-NAME PIC X(<i>n</i>).	1<= <i>n</i> <=255
VARCHAR(<i>n</i>)	Varying-length character string. For example, 01 VAR-NAME. 49 VAR-LEN PIC S9(4) USAGE BINARY. 49 VAR-TEXT PIC X(<i>n</i>).	The inner variables must have a level of 49.
GRAPHIC(<i>n</i>)	Fixed-length graphic string. For example, 01 VAR-NAME PIC G(<i>n</i>) USAGE IS DISPLAY-1.	<i>n</i> refers to the number of double-byte characters, not to the number of bytes. 1<= <i>n</i> <=127
VARGRAPHIC(<i>n</i>)	Varying-length graphic string. For example, 01 VAR-NAME. 49 VAR-LEN PIC S9(4) USAGE BINARY. 49 VAR-TEXT PIC G(<i>n</i>) USAGE IS DISPLAY-1.	<i>n</i> refers to the number of double-byte characters, not to the number of bytes. The inner variables must have a level of 49.
BINARY(<i>n</i>)	SQL TYPE IS BINARY(<i>n</i>)	1<= <i>n</i> <=255
VARBINARY(<i>n</i>)	SQL TYPE IS VARBINARY(<i>n</i>)	1<= <i>n</i> <=32 704
DATE	Fixed-length character string of length <i>n</i> . For example, 01 VAR-NAME PIC X(<i>n</i>).	If you are using a date exit routine, <i>n</i> is determined by that routine. Otherwise, <i>n</i> must be at least 10.
TIME	Fixed-length character string of length <i>n</i> . For example, 01 VAR-NAME PIC X(<i>n</i>).	If you are using a time exit routine, <i>n</i> is determined by that routine. Otherwise, <i>n</i> must be at least 6; to include seconds, <i>n</i> must be at least 8.
TIMESTAMP	Fixed-length character string of length <i>n</i> . For example, 01 VAR-NAME PIC X(<i>n</i>).	<i>n</i> must be at least 19. To include microseconds, <i>n</i> must be 26; if <i>n</i> is less than 26, truncation occurs on the microseconds part.
TIMESTAMP(0)	Fixed-length character string of length <i>n</i> . For example, 01 VAR-NAME PIC X(<i>n</i>).	<i>n</i> must be at least 19.
TIMESTAMP(<i>p</i>) <i>p</i> > 0	Fixed-length character string of length <i>n</i> . For example, 01 VAR-NAME PIC X(<i>n</i>)).	<i>n</i> must be at least 19. To include fractional seconds, <i>n</i> must be 20+ <i>x</i> where <i>x</i> is the number of fractional seconds to include; if <i>x</i> is less than <i>p</i> , truncation occurs on the fractional seconds part.
TIMESTAMP(0) WITH TIME ZONE	Varying-length character string. For example, 01 VAR-NAME. 49 VAR-LEN PIC S9(4) USAGE BINARY. 49 VAR-TEXT PIC X(<i>n</i>).	The inner variables must have a level of 49. <i>n</i> must be at least 25.

Table 66. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	COBOL host variable equivalent	Notes
TIMESTAMP(<i>p</i>) WITH TIME ZONE	Varying-length character string. For example, 01 VAR-NAME. 49 VAR-LEN PIC S9(4) USAGE BINARY. 49 VAR-TEXT PIC X(<i>n</i>).	The inner variables must have a level of 49. <i>n</i> must be at least 26+ <i>p</i> .
Result set locator	SQL TYPE IS RESULT-SET-LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	USAGE IS SQL TYPE IS BLOB-LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	USAGE IS SQL TYPE IS CLOB-LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	USAGE IS SQL TYPE IS DBCLOB-LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB(<i>n</i>)	USAGE IS SQL TYPE IS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
CLOB(<i>n</i>)	USAGE IS SQL TYPE IS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
DBCLOB(<i>n</i>)	USAGE IS SQL TYPE IS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823
XML	SQL TYPE IS XML AS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823
BLOB file reference	USAGE IS SQL TYPE IS BLOB-FILE	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB file reference	USAGE IS SQL TYPE IS CLOB-FILE	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB file reference	USAGE IS SQL TYPE IS DBCLOB-FILE	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
XML BLOB file reference	SQL TYPE IS XML AS BLOB-FILE	Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.
XML CLOB file reference	SQL TYPE IS XML AS CLOB-FILE	Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.

Table 66. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	COBOL host variable equivalent	Notes
XML DBCLOB file reference	SQL TYPE IS XML AS DBCLOB-FILE	Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.
ROWID	SQL TYPE IS ROWID	

Related concepts:

“Compatibility of SQL and language data types” on page 180

“LOB host variable, LOB locator, and LOB file reference variable declarations” on page 757

“Host variable data types for XML data in embedded SQL applications” on page 241

SQL statements in COBOL programs

You can code SQL statements in certain COBOL program sections.

The allowable sections are shown in the following table.

Table 67. Allowable SQL statements for COBOL program sections

SQL statement	Program section
BEGIN DECLARE SECTION END DECLARE SECTION	WORKING-STORAGE SECTION ¹ or LINKAGE SECTION
INCLUDE SQLCA	WORKING-STORAGE SECTION ¹ or LINKAGE SECTION
INCLUDE text-file-name	PROCEDURE DIVISION or DATA DIVISION ²
DECLARE TABLE DECLARE CURSOR	DATA DIVISION or PROCEDURE DIVISION
DECLARE VARIABLE	WORKING-STORAGE SECTION ¹
Other	PROCEDURE DIVISION

Notes:

1. If you use the DB2 coprocessor, you can use the LOCAL-STORAGE SECTION wherever WORKING-STORAGE SECTION is listed in the table.
2. When including host variable declarations, the INCLUDE statement must be in the WORKING-STORAGE SECTION or the LINKAGE SECTION.

You cannot put SQL statements in the DECLARATIVES section of a COBOL program.

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If you are using the DB2 precompiler, the EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines. If you are using the DB2 coprocessor, the EXEC and SQL keywords can be on different lines. Do not include any tokens between the two keywords EXEC and SQL except for COBOL comments, including debugging lines. Do not include SQL comments between the keywords EXEC and SQL.

If the SQL statement appears between two COBOL statements, the period after END-EXEC is optional and might not be appropriate. If the statement appears in an IF...THEN set of COBOL statements, omit the ending period to avoid inadvertently ending the IF statement.

You might code an UPDATE statement in a COBOL program as follows:

```
EXEC SQL
    UPDATE DSN8A10.DEPT
    SET MGRNO = :MGR-NUM
    WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Comments: You can include COBOL comment lines (* in column 7) in SQL statements wherever you can use a blank. If you are using the DB2 precompiler, you cannot include COBOL comment lines between the keywords EXEC and SQL. The precompiler treats COBOL debugging lines and page-eject lines (/ in column 7) as comment lines. The DB2 coprocessor treats the debugging lines based on the COBOL rules, which depend on the WITH DEBUGGING mode setting.

For an SQL INCLUDE statement, the DB2 precompiler treats any text that follows the period after END-EXEC, and on the same line as END-EXEC, as a comment. The DB2 coprocessor treats this text as part of the COBOL program syntax.

In addition, you can include SQL comments ('--') in any embedded SQL statement.

Debugging lines: The DB2 precompiler ignores the 'D' in column 7 on debugging lines and treats it as a blank. The DB2 coprocessor follows the COBOL language rules regarding debugging lines.

Continuation for SQL statements: The rules for continuing a character string constant from one line to the next in an SQL statement embedded in a COBOL program are the same as those for continuing a non-numeric literal in COBOL. However, you can use either a quote or an apostrophe as the first nonblank character in area B of the continuation line. The same rule applies for the continuation of delimited identifiers and does not depend on the string delimiter option.

To conform with SQL standard, delimit a character string constant with an apostrophe, and use a quote as the first nonblank character in area B of the continuation line for a character string constant.

Continued lines of an SQL statement can be in columns 8 through 72 when using the DB2 precompiler and columns 12 through 72 when using the DB2 coprocessor.

COPY: If you use the DB2 precompiler, do not use a COBOL COPY statement within host variable declarations. If you use the DB2 coprocessor, you can use COBOL COPY.

REPLACE: If you use the DB2 precompiler, the REPLACE statement has no effect on SQL statements. It affects only the COBOL statements that the precompiler generates.

If you use the DB2 coprocessor, the REPLACE statement replaces text strings in SQL statements as well as in generated COBOL statements.

Declaring tables and views: Your COBOL program should include the statement DECLARE TABLE to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. You should include the DCLGEN members in the DATA DIVISION.

Dynamic SQL in a COBOL program: In general, COBOL programs can easily handle dynamic SQL statements. COBOL programs can handle SELECT statements if the data types and the number of fields returned are fixed. If you want to use variable-list SELECT statements, use an SQLDA.

Including code: To include SQL statements or COBOL host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name END-EXEC.
```

If you are using the DB2 precompiler, you cannot nest SQL INCLUDE statements. In this case, do not use COBOL verbs to include SQL statements or host variable declarations, and do not use the SQL INCLUDE statement to include CICS preprocessor related code. In general, if you are using the DB2 precompiler, use the SQL INCLUDE statement only for SQL-related coding. If you are using the COBOL DB2 coprocessor, none of these restrictions apply.

Use the 'EXEC SQL' and 'END-EXEC' keyword pair to include SQL statements only. COBOL statements, such as COPY or REPLACE, are not allowed.

Margins: You must code SQL statements that begin with EXEC SQL in columns 12 through 72. Otherwise the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid COBOL name for a host variable. Do not use external entry names or access plan names that begin with 'DSN', and do not use host variable names that begin with 'SQL'. These names are reserved for DB2.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement labels: You can precede executable SQL statements in the PROCEDURE DIVISION with a paragraph name.

WHENEVER statement: The target for the GOTO clause in an SQL statement WHENEVER must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

Special COBOL considerations: The following considerations apply to programs written in COBOL:

- In a COBOL program that uses elements in a multi-level structure as host variable names, the DB2 precompiler generates the lowest two-level names.
- Using the COBOL compiler options DYNAM and NODYNAM depends on the operating environment.

TSO and IMS: You can specify the option DYNAM when compiling a COBOL program if you use the following guidelines. IMS and DB2 share a common alias name, DSNHLI, for the language interface module. You must do the following when you concatenate your libraries:

- If you use IMS with the COBOL option DYNAM, be sure to concatenate the IMS library first.

- If you run your application program only under DB2, be sure to concatenate the DB2 library first.

CICS, CAF, and RRSAF: You must specify the NODYNAM option when you compile a COBOL program that either includes CICS statements or is translated by a separate CICS translator or the integrated CICS translator. In these cases, you cannot specify the DYNAM option. If your CICS program has a subroutine that is not translated by a separate CICS translator or the integrated CICS translator but contains SQL statements, you can specify the DYNAM option. However, in this case, you must concatenate the CICS libraries before the DB2 libraries.

You can compile COBOL stored procedures with either the DYNAM option or the NODYNAM option. If you use DYNAM, ensure that the correct DB2 language interface module is loaded dynamically by performing one of the following actions:

- Use the ATTACH(RRSAF) precompiler option.
- Copy the DSNRLI module into a load library that is concatenated in front of the DB2 libraries. Use the member name DSNHLI.
- To avoid truncating numeric values, use either of the following methods:
 - Use the COMP-5 data type for binary integer host variables.
 - Specify the COBOL compiler option:
 - TRUNC(OPT) if you are certain that the data being moved to each binary variable by the application does not have a larger precision than is defined in the PICTURE clause of the binary variable.
 - TRUNC(BIN) if the precision of data being moved to each binary variable might exceed the value in the PICTURE clause.

DB2 assigns values to binary integer host variables as if you had specified the COBOL compiler option TRUNC(BIN) or used the COMP-5 data type.

- If you are using the DB2 precompiler and your COBOL program contains several entry points or is called several times, the USING clause of the entry statement that executes before the first SQL statement executes must contain the SQLCA and all linkage section entries that any SQL statement uses as host variables.
- If you use the DB2 precompiler, no compiler directives should appear between the PROCEDURE DIVISION and the DECLARATIVES statement.
- Do not use COBOL figurative constants (such as ZERO and SPACE), symbolic characters, reference modification, and subscripts within SQL statements.
- Observe the rules for naming SQL identifiers. However, for COBOL only, the names of SQL identifiers can follow the rules for naming COBOL words, if the names do not exceed the allowable length for the DB2 object. For example, the name 1ST-TIME is a valid cursor name because it is a valid COBOL word, but the name 1_TIME is not valid because it is not a valid SQL identifier or a valid COBOL word.
- Observe these rules for hyphens:
 - Surround hyphens used as subtraction operators with spaces. DB2 usually interprets a hyphen with no spaces around it as part of a host variable name.
 - You can use hyphens in SQL identifiers under either of the following circumstances:
 - The application program is a local application that runs on DB2 for z/OS Version 8 or later.
 - The application program accesses remote sites, and the local site and remote sites are DB2 for z/OS Version 8 or later.

- If you include an SQL statement in a COBOL PERFORM ... THRU paragraph and also specify the SQL statement WHENEVER ... GO, the COBOL compiler returns the warning message IGYOP3094. That message might indicate a problem. This usage is not recommended.
- If you are using the DB2 precompiler, all SQL statements and any host variables they reference must be within the first program when using nested programs or batch compilation.
- If you are using the DB2 precompiler, your COBOL programs must have a DATA DIVISION and a PROCEDURE DIVISION. Both divisions and the WORKING-STORAGE SECTION must be present in programs that contain SQL statements. However, if your COBOL programs requires the LOCAL-STORAGE SECTION, then the DB2 coprocessor should be used instead of the DB2 precompiler.

PSPI If your program uses the DB2 precompiler and uses parameters that are defined in LINKAGE SECTION as host variables to DB2 and the address of the input parameter might change on subsequent invocations of your program, your program must reset the variable SQL-INIT-FLAG. This flag is generated by the DB2 precompiler. Resetting this flag indicates that the storage must initialize when the next SQL statement executes. To reset the flag, insert the statement MOVE ZERO TO SQL-INIT-FLAG in the called program's PROCEDURE DIVISION, ahead of any executable SQL statements that use the host variables. If you use the COBOL DB2 coprocessor, the called program does not need to reset

SQL-INIT-FLAG. **PSPI**

You can use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR.

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program.

DSNTIAR syntax:

```
CALL 'DSNTIAR' USING sqlca message lrecl.
```

The DSNTIAR parameters have the following meanings:

sqlca

An SQL communication area.

message

An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
01 ERROR-MESSAGE.
    02 ERROR-LEN    PIC S9(4)  COMP VALUE +1320.
    02 ERROR-TEXT  PIC X(132) OCCURS 10 TIMES
                          INDEXED BY ERROR-INDEX.
77 ERROR-TEXT-LEN    PIC S9(9)  COMP VALUE +132.
```

```
⋮  
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

where ERROR-MESSAGE is the name of the message output area containing 10 lines of length 132 each, and ERROR-TEXT-LEN is the length of each line.

lrecl

A fullword containing the logical record length of output messages, between 72 and 240.

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSN8BC3, which is contained in the library DSN8A10.

CICS: If you call DSNTIAR dynamically from a CICS COBOL application program, be sure you do the following:

- Compile the COBOL application with the NODYNAM option.
- Define DSNTIAR in the CSD.

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL 'DSNTIAC' USING eib commarea sqlca msg lrecl.
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

eib EXEC interface block

commarea
communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIAC1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix.SDSNSAMP*.

Related concepts:

“DB2 sample applications” on page 1092

“DCLGEN (declarations generator)” on page 161

“Host variable arrays in an SQL statement” on page 191

 SQL identifiers (DB2 SQL)

Related tasks:

“Including dynamic SQL in your program” on page 193

“Embedding SQL statements in your application” on page 183

“Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234

“Defining SQL descriptor areas” on page 173

“Displaying SQLCA fields by calling DSNTIAR” on page 229

☞ Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Delimiters in SQL statements in COBOL programs

You must delimit SQL statements in your COBOL program so that DB2 knows when a particular SQL statement ends.

Delimit an SQL statement in your COBOL program with the beginning keyword EXEC SQL and an END-EXEC.

Example

Use EXEC SQL and END-EXEC. to delimit an SQL statement in a COBOL program:

```
EXEC SQL
    an SQL statement
END-EXEC.
```

Object-oriented extensions in COBOL

When you use object-oriented extensions in a COBOL application, you need to consider where to place SQL statements, the SQLCA, the SQLDA, and host variable declarations. You also need to consider the rules for host variables.

Where to place SQL statements in your application: A COBOL source data set or member can contain the following elements:

- Multiple programs
- Multiple class definitions, each of which contains multiple methods

You can put SQL statements in only the first program or class in the source data set or member. However, you can put SQL statements in multiple methods within a class. If an application consists of multiple data sets or members, each of the data sets or members can contain SQL statements.

Where to place the SQLCA, SQLDA, and host variable declarations: You can put the SQLCA, SQLDA, and SQL host variable declarations in the WORKING-STORAGE SECTION of a program, class, or method. An SQLCA or SQLDA in a class WORKING-STORAGE SECTION is global for all the methods of the class. An SQLCA or SQLDA in a method WORKING-STORAGE SECTION is local to that method only.

If a class and a method within the class both contain an SQLCA or SQLDA, the method uses the SQLCA or SQLDA that is local.

Rules for host variables: You can declare COBOL variables that are used as host variables in the WORKING-STORAGE SECTION or LINKAGE-SECTION of a program, class, or method. You can also declare host variables in the LOCAL-STORAGE SECTION of a method. The scope of a host variable is the method, class, or program within which it is defined.

Programming examples in COBOL

You can write DB2 programs in COBOL. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in DSN910.SDSNSAMP as a model for your JCL.

Related reference:

“Programming examples” on page 251

Sample COBOL dynamic SQL program

You can code dynamic varying-list SELECT statements in a COBOL program. *Varying-List SELECT statements* are statements for which you do not know the number or data types of columns that are to be returned when you write the program.

Introductory concepts:

Dynamic SQL applications (Introduction to DB2 for z/OS)

“Including dynamic SQL in your program” on page 193 describes three variations of dynamic SQL statements:

- Non-SELECT statements
- Fixed-List SELECT statements

In this case, you know the number of columns returned and their data types when you write the program.

- Varying-List SELECT statements.

In this case, you do **not** know the number of columns returned and their data types when you write the program.

This section documents a technique of coding varying list SELECT statements in COBOL.

This example program does not support BLOB, CLOB, or DBCLOB data types.

Pointers and based variables in the sample COBOL program

COBOL has a POINTER type and a SET statement that provide pointers and based variables.

The SET statement sets a pointer from the address of an area in the linkage section or another pointer; the statement can also set the address of an area in the linkage section. UNLDBC2 in “Example of the sample COBOL program” on page 365 provides these uses of the SET statement. The SET statement does not permit the use of an address in the WORKING-STORAGE section.

Storage allocation for the sample COBOL program

COBOL does not provide a means to allocate main storage within a program. You can achieve the same end by having an initial program which allocates the storage, and then calls a second program that manipulates the pointer. (COBOL does not permit you to directly manipulate the pointer because errors and abends are likely to occur.)

The initial program is extremely simple. It includes a working storage section that allocates the maximum amount of storage needed. This program then calls the second program, passing the area or areas on the CALL statement. The second program defines the area in the linkage section and can then use pointers within the area.

If you need to allocate parts of storage, the best method is to use indexes or subscripts. You can use subscripts for arithmetic and comparison operations.

Example of the sample COBOL program

The following example shows an example of the initial program UNLDBCUI1 that allocates the storage and calls the second program UNLDBCUI2. UNLDBCUI2 then defines the passed storage areas in its linkage section and includes the USING clause on its PROCEDURE DIVISION statement.

Defining the pointers, then redefining them as numeric, permits some manipulation of the pointers that you cannot perform directly. For example, you cannot add the column length to the record pointer, but you can add the column length to the numeric value that redefines the pointer.

The following example is the initial program that allocates storage.

```

**** UNLDBCUI1- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM *****
*
*   MODULE NAME = UNLDBCUI1
*
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION
*                       UNLOAD PROGRAM
*                       BATCH
*                       IBM ENTERPRISE COBOL FOR Z/OS
*
*   COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1987
*   REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
*
*   STATUS = VERSION 1 RELEASE 3, LEVEL 0
*
*   FUNCTION = THIS MODULE PROVIDES THE STORAGE NEEDED BY
*               UNLDBCUI2 AND CALLS THAT PROGRAM.
*
*   NOTES =
*   DEPENDENCIES = ENTERPRISE COBOL FOR Z/OS IS REQUIRED.
*                   SEVERAL NEW FACILITIES ARE USED.
*
*   RESTRICTIONS =
*   THE MAXIMUM NUMBER OF COLUMNS IS 750,
*   WHICH IS THE SQL LIMIT.
*
*   DATA RECORDS ARE LIMITED TO 32700 BYTES,
*   INCLUDING DATA, LENGTHS FOR VARCHAR DATA,
*   AND SPACE FOR NULL INDICATORS.
*
*   MODULE TYPE = IBM ENTERPRISE COBOL PROGRAM
*   PROCESSOR   = ENTERPRISE COBOL FOR Z/OS
*   MODULE SIZE = SEE LINK EDIT
*   ATTRIBUTES  = REENTRANT
*
*   ENTRY POINT = UNLDBCUI1
*   PURPOSE     = SEE FUNCTION
*   LINKAGE     = INVOKED FROM DSN RUN
*   INPUT      = NONE
*   OUTPUT     = NONE
*
*   EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION
*
*   EXIT-ERROR =
*   RETURN CODE = NONE
*   ABEND CODES = NONE
*   ERROR-MESSAGES = NONE
*
*   EXTERNAL REFERENCES =

```



```

*      ROUTINES/SERVICES =
*          UNLDBC2 - ACTUAL UNLOAD PROGRAM
*
*      DATA-AREAS      = NONE
*      CONTROL-BLOCKS  = NONE
*
*      TABLES = NONE
*      CHANGE-ACTIVITY = NONE
*
* *PSEUDOCODE*
*
*      PROCEDURE
*      CALL UNLDBC2.
*      END.
*-----*
/
IDENTIFICATION DIVISION.
*-----
PROGRAM-ID.    UNLDBC1
*
ENVIRONMENT DIVISION.
*
CONFIGURATION SECTION.
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
01  WORKAREA-IND.
    02  WORKIND PIC S9(4) COMP OCCURS 750 TIMES.
01  RECWORK.
    02  RECWORK-LEN PIC S9(8) COMP VALUE 32700.
    02  RECWORK-CHAR PIC X(1) OCCURS 32700 TIMES.
*
PROCEDURE DIVISION.
*
    CALL 'UNLDBC2' USING WORKAREA-IND RECWORK.
    GOBACK.

```

The following example is the called program that does pointer manipulation.

```

**** UNLDBC2- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM *****
*
*  MODULE NAME = UNLDBC2
*
*  DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION
*                    UNLOAD PROGRAM
*                    BATCH
*                    ENTERPRISE COBOL FOR Z/OS
*
*  COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1987
*  REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
*
*  STATUS = VERSION 1 RELEASE 3, LEVEL 0
*
*  FUNCTION = THIS MODULE ACCEPTS A TABLE NAME OR VIEW NAME
*            AND UNLOADS THE DATA IN THAT TABLE OR VIEW.
*  READ IN A TABLE NAME FROM SYSIN.
*  PUT DATA FROM THE TABLE INTO DD SYSRECO1.
*  WRITE RESULTS TO SYSPRINT.
*
*  NOTES =
*  DEPENDENCIES = IBM ENTERPRISE COBOL FOR Z/OS
*                IS REQUIRED.
*
*  RESTRICTIONS =
*  THE SQLDA IS LIMITED TO 33016 BYTES.
*  THIS SIZE ALLOWS FOR THE DB2 MAXIMUM
*

```



```

*           OF 750 COLUMNS.
*
*           DATA RECORDS ARE LIMITED TO 32700 BYTES,
*           INCLUDING DATA, LENGTHS FOR VARCHAR DATA,
*           AND SPACE FOR NULL INDICATORS.
*
*           TABLE OR VIEW NAMES ARE ACCEPTED, AND ONLY
*           ONE NAME IS ALLOWED PER RUN.
*
* MODULE TYPE = ENTERPRISE COBOL FOR Z/OS
*   PROCESSOR = DB2 PRECOMPILER, COBOL COMPILER
*   MODULE SIZE = SEE LINK EDIT
*   ATTRIBUTES = REENTRANT
*
* ENTRY POINT = UNLDBC2
*   PURPOSE = SEE FUNCTION
*   LINKAGE =
*     CALL 'UNLDBC2' USING WORKAREA-IND REWORK.
*
*   INPUT = SYMBOLIC LABEL/NAME = WORKAREA-IND
*           DESCRIPTION = INDICATOR VARIABLE ARRAY
*           01 WORKAREA-IND.
*           02 WORKIND PIC S9(4) COMP OCCURS 750 TIMES.
*
*           SYMBOLIC LABEL/NAME = REWORK
*           DESCRIPTION = WORK AREA FOR OUTPUT RECORD
*           01 REWORK.
*           02 REWORK-LEN PIC S9(8) COMP.
*           02 REWORK-CHAR PIC X(1) OCCURS 32700 TIMES.
*
*           SYMBOLIC LABEL/NAME = SYSIN
*           DESCRIPTION = INPUT REQUESTS - TABLE OR VIEW
*
*   OUTPUT = SYMBOLIC LABEL/NAME = SYSPRINT
*            DESCRIPTION = PRINTED RESULTS
*
*           SYMBOLIC LABEL/NAME = SYSREC01
*           DESCRIPTION = UNLOADED TABLE DATA
*
* EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION
* EXIT-ERROR =
*   RETURN CODE = NONE
*   ABEND CODES = NONE
*   ERROR-MESSAGES =
*     DSNT490I SAMPLE COBOL DATA UNLOAD PROGRAM RELEASE 3.0*
*       - THIS IS THE HEADER, INDICATING A NORMAL
*       - START FOR THIS PROGRAM.
*     DSNT493I SQL ERROR, SQLCODE = NNNNNNNN
*       - AN SQL ERROR OR WARNING WAS ENCOUNTERED
*       - ADDITIONAL INFORMATION FROM DSNTIAR
*       - FOLLOWS THIS MESSAGE.
*     DSNT495I SUCCESSFUL UNLOAD XXXXXXXX ROWS OF
*       TABLE TTTTTTTT
*       - THE UNLOAD WAS SUCCESSFUL. XXXXXXXX IS
*       - THE NUMBER OF ROWS UNLOADED. TTTTTTTT
*       - IS THE NAME OF THE TABLE OR VIEW FROM
*       - WHICH IT WAS UNLOADED.
*     DSNT496I UNRECOGNIZED DATA TYPE CODE OF NNNNN
*       - THE PREPARE RETURNED AN INVALID DATA
*       - TYPE CODE. NNNNN IS THE CODE, PRINTED
*       - IN DECIMAL. USUALLY AN ERROR IN
*       - THIS ROUTINE OR A NEW DATA TYPE.
*     DSNT497I RETURN CODE FROM MESSAGE ROUTINE DSNTIAR
*       - THE MESSAGE FORMATTING ROUTINE DETECTED
*       - AN ERROR. SEE THAT ROUTINE FOR RETURN
*       - CODE INFORMATION. USUALLY AN ERROR IN
*       - THIS ROUTINE.

```

```

*          DSNT498I ERROR, NO VALID COLUMNS FOUND          *
*          - THE PREPARE RETURNED DATA WHICH DID NOT      *
*          - PRODUCE A VALID OUTPUT RECORD.                *
*          - USUALLY AN ERROR IN THIS ROUTINE.             *
*          DSNT499I NO ROWS FOUND IN TABLE OR VIEW         *
*          - THE CHOSEN TABLE OR VIEWS DID NOT            *
*          - RETURN ANY ROWS.                               *
*          ERROR MESSAGES FROM MODULE DSNTIAR              *
*          - WHEN AN ERROR OCCURS, THIS MODULE             *
*          - PRODUCES CORRESPONDING MESSAGES.              *
*          OTHER MESSAGES:                                  *
*          THE TABLE COULD NOT BE UNLOADED. EXITING.      *
*
*          EXTERNAL REFERENCES =                            *
*          ROUTINES/SERVICES =                             *
*          DSNTIAR - TRANSLATE SQLCA INTO MESSAGES         *
*          DATA-AREAS = NONE                              *
*          CONTROL-BLOCKS =                                *
*          SQLCA - SQL COMMUNICATION AREA                  *
*
*          TABLES = NONE                                   *
*          CHANGE-ACTIVITY = NONE                          *
*
*          *PSEUDOCODE*                                     *
*          PROCEDURE                                        *
*          EXEC SQL DECLARE DT CURSOR FOR SEL END-EXEC.     *
*          EXEC SQL DECLARE SEL STATEMENT END-EXEC.        *
*          INITIALIZE THE DATA, OPEN FILES.               *
*          OBTAIN STORAGE FOR THE SQLDA AND THE DATA RECORDS. *
*          READ A TABLE NAME.                              *
*          OPEN SYSREC01.                                    *
*          BUILD THE SQL STATEMENT TO BE EXECUTED          *
*          EXEC SQL PREPARE SQL STATEMENT INTO SQLDA END-EXEC. *
*          SET UP ADDRESSES IN THE SQLDA FOR DATA.         *
*          INITIALIZE DATA RECORD COUNTER TO 0.           *
*          EXEC SQL OPEN DT END-EXEC.                       *
*          DO WHILE SQLCODE IS 0.                           *
*          EXEC SQL FETCH DT USING DESCRIPTOR SQLDA END-EXEC. *
*          ADD IN MARKERS TO DENOTE NULLS.                 *
*          WRITE THE DATA TO SYSREC01.                    *
*          INCREMENT DATA RECORD COUNTER.                 *
*          END.                                              *
*          EXEC SQL CLOSE DT END-EXEC.                     *
*          INDICATE THE RESULTS OF THE UNLOAD OPERATION.   *
*          CLOSE THE SYSIN, SYSPRINT, AND SYSREC01 FILES. *
*          END.                                              *
*-----*
/
IDENTIFICATION DIVISION.
*-----*
PROGRAM-ID. UNLDBC2
*
ENVIRONMENT DIVISION.
*-----*
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SYSIN
        ASSIGN TO DA-S-SYSIN.
    SELECT SYSPRINT
        ASSIGN TO UT-S-SYSPRINT.
    SELECT SYSREC01
        ASSIGN TO DA-S-SYSREC01.
*
DATA DIVISION.
*-----*
*

```

```

FILE SECTION.
FD      SYSIN
        RECORD CONTAINS 80 CHARACTERS
        BLOCK CONTAINS 0 RECORDS
        LABEL RECORDS ARE OMITTED
        RECORDING MODE IS F.
01 CARDREC          PIC X(80).
*
FD      SYSPRINT
        RECORD CONTAINS 120 CHARACTERS
        LABEL RECORDS ARE OMITTED
        DATA RECORD IS MSGREC
        RECORDING MODE IS F.
01 MSGREC          PIC X(120).
*
FD      SYSREC01
        RECORD CONTAINS 5 TO 32704 CHARACTERS
        LABEL RECORDS ARE OMITTED
        DATA RECORD IS REC01
        RECORDING MODE IS V.
01 REC01.
    02 REC01-LEN PIC S9(8) COMP.
    02 REC01-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
        DEPENDING ON REC01-LEN.
/
WORKING-STORAGE SECTION.
*
*****
* STRUCTURE FOR INPUT
*****
01 IOAREA.
    02 TNAME          PIC X(72).
    02 FILLER        PIC X(08).
01 STMTBUF.
    49 STMTLEN       PIC S9(4) COMP VALUE 92.
    49 STMTCHAR      PIC X(92).
01 STMTBLD.
    02 FILLER        PIC X(20) VALUE 'SELECT * FROM'.
    02 STMTTAB       PIC X(72).
*
*****
* REPORT HEADER STRUCTURE
*****
01 HEADER.
    02 FILLER PIC X(35)
        VALUE ' DSNT490I SAMPLE COBOL DATA UNLOAD '.
    02 FILLER PIC X(85) VALUE 'PROGRAM RELEASE 3.0'.
01 MSG-SQLERR.
    02 FILLER PIC X(31)
        VALUE ' DSNT493I SQL ERROR, SQLCODE = '.
    02 MSG-MINUS      PIC X(1).
    02 MSG-PRINT-CODE PIC 9(8).
    02 FILLER PIC X(81) VALUE ' '.
01 MSG-OTHER-ERR.
    02 FILLER PIC X(42)
        VALUE ' THE TABLE COULD NOT BE UNLOADED. EXITING.'.
    02 FILLER PIC X(78) VALUE ' '.
01 UNLOADED.
    02 FILLER PIC X(28)
        VALUE ' DSNT495I SUCCESSFUL UNLOAD '.
    02 ROWS          PIC 9(8).
    02 FILLER PIC X(15) VALUE ' ROWS OF TABLE '.
    02 TABLENAM      PIC X(72) VALUE ' '.
01 BADTYPE.
    02 FILLER PIC X(42)
        VALUE ' DSNT496I UNRECOGNIZED DATA TYPE CODE OF '.
    02 TYPCOD        PIC 9(8).

```

```

02 FILLER PIC X(71) VALUE ' '.
01 MSGRETCO.
02 FILLER PIC X(42)
VALUE ' DSNT497I RETURN CODE FROM MESSAGE ROUTINE'.
02 FILLER PIC X(9) VALUE 'DSNTIAR '.
02 RETCODE PIC 9(8).
02 FILLER PIC X(62) VALUE ' '.
01 MSGNOCOL.
02 FILLER PIC X(120)
VALUE ' DSNT498I ERROR, NO VALID COLUMNS FOUND'.
01 MSG-NOROW.
02 FILLER PIC X(120)
VALUE ' DSNT499I NO ROWS FOUND IN TABLE OR VIEW'.
*****
* WORKAREAS *
*****
77 NOT-FOUND PIC S9(8) COMP VALUE +100.
*****
* VARIABLES FOR ERROR-MESSAGE FORMATTING *
*****
01 ERROR-MESSAGE.
02 ERROR-LEN PIC S9(4) COMP VALUE +960.
02 ERROR-TEXT PIC X(120) OCCURS 8 TIMES
INDEXED BY ERROR-INDEX.
77 ERROR-TEXT-LEN PIC S9(8) COMP VALUE +120.
*****
* SQL DESCRIPTOR AREA *
*****
01 SQLDA.
02 SQLDAID PIC X(8) VALUE 'SQLDA '.
02 SQLDABC PIC S9(8) COMPUTATIONAL VALUE 33016.
02 SQLN PIC S9(4) COMPUTATIONAL VALUE 750.
02 SQLD PIC S9(4) COMPUTATIONAL VALUE 0.
02 SQLVAR OCCURS 1 TO 750 TIMES
DEPENDING ON SQLN.
03 SQLTYPE PIC S9(4) COMPUTATIONAL.
03 SQLLEN PIC S9(4) COMPUTATIONAL.
03 SQLDATA POINTER.
03 SQLIND POINTER.
03 SQLNAME.
49 SQLNAMEL PIC S9(4) COMPUTATIONAL.
49 SQLNAMEC PIC X(30).
*
* DATA TYPES FOUND IN SQLTYPE, AFTER REMOVING THE NULL BIT
*
77 VARCTYPE PIC S9(4) COMP VALUE +448.
77 CHARTYPE PIC S9(4) COMP VALUE +452.
77 VARLTYPE PIC S9(4) COMP VALUE +456.
77 VARGTYPE PIC S9(4) COMP VALUE +464.
77 GTYPE PIC S9(4) COMP VALUE +468.
77 LVARGTYP PIC S9(4) COMP VALUE +472.
77 FLOATYPE PIC S9(4) COMP VALUE +480.
77 DECTYPE PIC S9(4) COMP VALUE +484.
77 INTTYPE PIC S9(4) COMP VALUE +496.
77 HWTYPE PIC S9(4) COMP VALUE +500.
77 DATETYP PIC S9(4) COMP VALUE +384.
77 TIMETYP PIC S9(4) COMP VALUE +388.
77 TIMESTMP PIC S9(4) COMP VALUE +392.
*
01 RECPTR POINTER.
01 RECNUM REDEFINES RECPTR PICTURE S9(8) COMPUTATIONAL.
01 IRECPTR POINTER.
01 IRECNUM REDEFINES IRECPTR PICTURE S9(8) COMPUTATIONAL.
01 I PICTURE S9(4) COMPUTATIONAL.
01 J PICTURE S9(4) COMPUTATIONAL.
01 DUMMY PICTURE S9(4) COMPUTATIONAL.
01 MYTYPE PICTURE S9(4) COMPUTATIONAL.

```

```

01 COLUMN-IND PICTURE S9(4) COMPUTATIONAL.
01 COLUMN-LEN PICTURE S9(4) COMPUTATIONAL.
01 COLUMN-PREC PICTURE S9(4) COMPUTATIONAL.
01 COLUMN-SCALE PICTURE S9(4) COMPUTATIONAL.
01 INDCOUNT          PIC S9(4) COMPUTATIONAL.
01 ROWCOUNT        PIC S9(4) COMPUTATIONAL.
01 ERR-FOUND PICTURE X(1).
01 WORKAREA2.
    02 WORKINDPTR POINTER OCCURS 750 TIMES.
*****
* DECLARE CURSOR AND STATEMENT FOR DYNAMIC SQL
*****
*
    EXEC SQL DECLARE DT CURSOR FOR SEL END-EXEC.
    EXEC SQL DECLARE SEL STATEMENT END-EXEC.
*
*****
* SQL INCLUDE FOR SQLCA
*****
    EXEC SQL INCLUDE SQLCA END-EXEC.
*
77 ONE          PIC S9(4) COMP VALUE +1.
77 TWO         PIC S9(4) COMP VALUE +2.
77 FOUR        PIC S9(4) COMP VALUE +4.
77 QMARK       PIC X(1) VALUE '?'.
*
LINKAGE SECTION.
01 LINKAREA-IND.
    02 IND PIC S9(4) COMP OCCURS 750 TIMES.
01 LINKAREA-REC.
    02 REC1-LEN PIC S9(8) COMP.
    02 REC1-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
        DEPENDING ON REC1-LEN.
01 LINKAREA-QMARK.
    02 INDREC PIC X(1).
/
PROCEDURE DIVISION USING LINKAREA-IND LINKAREA-REC.
*
*****
* SQL RETURN CODE HANDLING
*****
    EXEC SQL WHENEVER SQLERROR GOTO DBERROR END-EXEC.
    EXEC SQL WHENEVER SQLWARNING GOTO DBERROR END-EXEC.
    EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
*
*****
* MAIN PROGRAM ROUTINE
*****
    SET IRECPTR TO ADDRESS OF REC1-CHAR(1).
*
    MOVE 'N' TO ERR-FOUND.
*
    OPEN INPUT SYSIN
*
    OUTPUT SYSPRINT
    OUTPUT SYSREC01.
*
    WRITE MSGREC FROM HEADER
        AFTER ADVANCING 2 LINES.
*
    READ SYSIN RECORD INTO IOAREA.
*
    PERFORM PROCESS-INPUT THROUGH IND-RESULT.
*
    PROG-END.
*
**OPEN FILES
**INITIALIZE
** ERROR FLAG
**WRITE HEADER
**GET FIRST INPUT
**MAIN ROUTINE
**CLOSE FILES

```



```

*
*   INDICATE THE RESULTS OF THE UNLOAD OPERATION.
*
*
*****
IND-RESULT.
  IF ERR-FOUND = 'N' THEN
    MOVE TNAME TO TABLENAM
    MOVE ROWCOUNT TO ROWS
    WRITE MSGREC FROM UNLOADED
      AFTER ADVANCING 2 LINES
  ELSE
    WRITE MSGREC FROM MSG-OTHER-ERR
      AFTER ADVANCING 2 LINES
    MOVE +0012 TO RETURN-CODE
    GO TO PROG-END.
*
WRITE-AND-FETCH.
*   ADD IN MARKERS TO DENOTE NULLS.
  MOVE ONE TO INDCOUNT.
  PERFORM NULLCHK UNTIL INDCOUNT = SQLD.
  MOVE REC1-LEN TO REC01-LEN.
  WRITE REC01 FROM LINKAREA-REC.
  ADD ONE TO ROWCOUNT.
  PERFORM BLANK-REC.
  EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
*
NULLCHK.
  IF IND(INDCOUNT) < 0 THEN
    SET ADDRESS OF LINKAREA-QMARK TO WORKINDPTR(INDCOUNT)
    MOVE QMARK TO INDREC.
    ADD ONE TO INDCOUNT.
*****
*   BLANK OUT RECORD TEXT FIRST
*
*****
BLANK-REC.
  MOVE ONE TO J.
  PERFORM BLANK-MORE UNTIL J > REC1-LEN.
BLANK-MORE.
  MOVE ' ' TO REC1-CHAR(J).
  ADD ONE TO J.
*
COLADDR.
  SET SQLDATA(I) TO RECPTR.
*****
*
*   DETERMINE THE LENGTH OF THIS COLUMN (COLUMN-LEN)
*   THIS DEPENDS UPON THE DATA TYPE.  MOST DATA TYPES HAVE
*   THE LENGTH SET, BUT VARCHAR, GRAPHIC, VARGRAPHIC, AND
*   DECIMAL DATA NEED TO HAVE THE BYTES CALCULATED.
*   THE NULL ATTRIBUTE MUST BE SEPARATED TO SIMPLIFY MATTERS.
*
*****
  MOVE SQLLEN(I) TO COLUMN-LEN.
*   COLUMN-IND IS 0 FOR NO NULLS AND 1 FOR NULLS
  DIVIDE SQLTYPE(I) BY TWO GIVING DUMMY REMAINDER COLUMN-IND.
*   MYTYPE IS JUST THE SQLTYPE WITHOUT THE NULL BIT
  MOVE SQLTYPE(I) TO MYTYPE.
  SUBTRACT COLUMN-IND FROM MYTYPE.
*   SET THE COLUMN LENGTH, DEPENDENT UPON DATA TYPE
  EVALUATE MYTYPE
    WHEN CHARTYPE CONTINUE,
    WHEN DATETYP CONTINUE,
    WHEN TIMETYP CONTINUE,
    WHEN TIMESTMP CONTINUE,
    WHEN FLOATYPE CONTINUE,
    WHEN VARCTYPE
      ADD TWO TO COLUMN-LEN,

```



```

*          SYMBOLIC LABEL/NAME = SYSPRINT          *
*          DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH *
*          STEP AND THE RESULTANT SQLCA          *
*
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION *
*
* EXIT ERROR = NONE *
*
* EXTERNAL REFERENCES = *
*   ROUTINE SERVICES = NONE *
*   DATA-AREAS      = NONE *
*   CONTROL-BLOCKS  = *
*   SQLCA - SQL COMMUNICATION AREA *
*
* TABLES = NONE *
*
* CHANGE-ACTIVITY = NONE *
*
*
*
* PSEUDOCODE *
*
* MAINLINE. *
*   Perform CONNECT-TO-SITE-1 to establish *
*   a connection to the local connection. *
*   If the previous operation was successful Then *
*   Do. *
*     Perform PROCESS-CURSOR-SITE-1 to obtain the *
*     information about an employee that is *
*     transferring to another location. *
*     If the information about the employee was obtained *
*     successfully Then *
*     Do. *
*       Perform UPDATE-ADDRESS to update the information *
*       to contain current information about the *
*       employee. *
*       Perform CONNECT-TO-SITE-2 to establish *
*       a connection to the site where the employee is *
*       transferring to. *
*       If the connection is established successfully *
*       Then *
*       Do. *
*         Perform PROCESS-SITE-2 to insert the *
*         employee information at the location *
*         where the employee is transferring to. *
*         End if the connection was established *
*         successfully. *
*       End if the employee information was obtained *
*       successfully. *
*     End if the previous operation was successful. *
*   Perform COMMIT-WORK to COMMIT the changes made to STLEC1 *
*   and STLEC2. *
*
* PROG-END. *
*   Close the printer. *
*   Return. *
*
* CONNECT-TO-SITE-1. *
*   Provide a text description of the following step. *
*   Establish a connection to the location where the *
*   employee is transferring from. *
*   Print the SQLCA out. *
*
* PROCESS-CURSOR-SITE-1. *
*   Provide a text description of the following step. *
*   Open a cursor that will be used to retrieve information *
*   about the transferring employee from this site. *

```

```

*      Print the SQLCA out.
*      If the cursor was opened successfully Then
*          Do.
*              Perform FETCH-DELETE-SITE-1 to retrieve and
*              delete the information about the transferring
*              employee from this site.
*              Perform CLOSE-CURSOR-SITE-1 to close the cursor.
*          End if the cursor was opened successfully.
*
*
*      FETCH-DELETE-SITE-1.
*          Provide a text description of the following step.
*          Fetch information about the transferring employee.
*          Print the SQLCA out.
*          If the information was retrieved successfully Then
*              Do.
*                  Perform DELETE-SITE-1 to delete the employee
*                  at this site.
*              End if the information was retrieved successfully.
*
*
*      DELETE-SITE-1.
*          Provide a text description of the following step.
*          Delete the information about the transferring employee
*          from this site.
*          Print the SQLCA out.
*
*
*      CLOSE-CURSOR-SITE-1.
*          Provide a text description of the following step.
*          Close the cursor used to retrieve information about
*          the transferring employee.
*          Print the SQLCA out.
*
*
*      UPDATE-ADDRESS.
*          Update the address of the employee.
*          Update the city of the employee.
*          Update the location of the employee.
*
*
*      CONNECT-TO-SITE-2.
*          Provide a text description of the following step.
*          Establish a connection to the location where the
*          employee is transferring to.
*          Print the SQLCA out.
*
*
*      PROCESS-SITE-2.
*          Provide a text description of the following step.
*          Insert the employee information at the location where
*          the employee is being transferred to.
*          Print the SQLCA out.
*
*
*      COMMIT-WORK.
*          COMMIT all the changes made to STLEC1 and STLEC2.
*
*
*****

```

```

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PRINTER, ASSIGN TO S-OUT1.

```

```

DATA DIVISION.
FILE SECTION.
FD PRINTER
RECORD CONTAINS 120 CHARACTERS
DATA RECORD IS PRT-TC-RESULTS
LABEL RECORD IS OMITTED.
01 PRT-TC-RESULTS.
03 PRT-BLANK PIC X(120).

```

WORKING-STORAGE SECTION.

* Variable declarations *

```
01 H-EMPTBL.  
  05 H-EMPNO   PIC X(6).  
  05 H-NAME.  
     49 H-NAME-LN   PIC S9(4) COMP-4.  
     49 H-NAME-DA   PIC X(32).  
  05 H-ADDRESS.  
     49 H-ADDRESS-LN PIC S9(4) COMP-4.  
     49 H-ADDRESS-DA PIC X(36).  
  05 H-CITY.  
     49 H-CITY-LN   PIC S9(4) COMP-4.  
     49 H-CITY-DA   PIC X(36).  
  05 H-EMPLOC  PIC X(4).  
  05 H-SSNO   PIC X(11).  
  05 H-BORN   PIC X(10).  
  05 H-SEX    PIC X(1).  
  05 H-HIRED  PIC X(10).  
  05 H-DEPTNO PIC X(3).  
  05 H-JOBCODE PIC S9(3)V COMP-3.  
  05 H-SRATE  PIC S9(5) COMP.  
  05 H-EDUC   PIC S9(5) COMP.  
  05 H-SAL    PIC S9(6)V9(2) COMP-3.  
  05 H-VALIDCHK PIC S9(6)V COMP-3.
```

```
01 H-EMPTBL-IND-TABLE.  
  02 H-EMPTBL-IND          PIC S9(4) COMP OCCURS 15 TIMES.
```

* Includes for the variables used in the COBOL standard *
* language procedures and the SQLCA. *

```
EXEC SQL INCLUDE COBSVAR END-EXEC.  
EXEC SQL INCLUDE SQLCA END-EXEC.
```

* Declaration for the table that contains employee information *

```
EXEC SQL DECLARE SYSADM.EMP TABLE  
  (EMPNO CHAR(6) NOT NULL,  
   NAME  VARCHAR(32),  
   ADDRESS VARCHAR(36) ,  
   CITY  VARCHAR(36) ,  
   EMPLOC CHAR(4) NOT NULL,  
   SSNO  CHAR(11),  
   BORN  DATE,  
   SEX  CHAR(1),  
   HIRED CHAR(10),  
   DEPTNO CHAR(3) NOT NULL,  
   JOBCODE DECIMAL(3),  
   SRATE  SMALLINT,  
   EDUC  SMALLINT,  
   SAL   DECIMAL(8,2) NOT NULL,  
   VALCHK DECIMAL(6))  
END-EXEC.
```

* Constants *

```
77 SITE-1          PIC X(16) VALUE 'STLEC1'.
```

```

77 SITE-2                PIC X(16) VALUE 'STLEC2'.
77 TEMP-EMPNO            PIC X(6)  VALUE '080000'.
77 TEMP-ADDRESS-LN      PIC 99   VALUE 15.
77 TEMP-CITY-LN         PIC 99   VALUE 18.

*****
* Declaration of the cursor that will be used to retrieve *
* information about a transferring employee                *
*****

      EXEC SQL DECLARE C1 CURSOR FOR
          SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
                 SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
                 SRATE, EDUC, SAL, VALCHK
          FROM   SYSADM.EMP
          WHERE  EMPNO = :TEMP-EMPNO
      END-EXEC.

PROCEDURE DIVISION.
A101-HOUSE-KEEPING.
      OPEN OUTPUT PRINTER.

*****
* An employee is transferring from location STLEC1 to STLEC2. *
* Retrieve information about the employee from STLEC1, delete *
* the employee from STLEC1 and insert the employee at STLEC2 *
* using the information obtained from STLEC1.                 *
*****

MAINLINE.
      PERFORM CONNECT-TO-SITE-1
      IF SQLCODE IS EQUAL TO 0
          PERFORM PROCESS-CURSOR-SITE-1
          IF SQLCODE IS EQUAL TO 0
              PERFORM UPDATE-ADDRESS
              PERFORM CONNECT-TO-SITE-2
              IF SQLCODE IS EQUAL TO 0
                  PERFORM PROCESS-SITE-2.
      PERFORM COMMIT-WORK.

PROG-END.
      CLOSE PRINTER.
      GOBACK.

*****
* Establish a connection to STLEC1                            *
*****

CONNECT-TO-SITE-1.

      MOVE 'CONNECT TO STLEC1 ' TO STNAME
      WRITE PRT-TC-RESULTS FROM STNAME
      EXEC SQL
          CONNECT TO :SITE-1
      END-EXEC.
      PERFORM PTSQLCA.

*****
* When a connection has been established successfully at STLEC1,*
* open the cursor that will be used to retrieve information    *
* about the transferring employee.                             *
*****

PROCESS-CURSOR-SITE-1.

      MOVE 'OPEN CURSOR C1      ' TO STNAME
      WRITE PRT-TC-RESULTS FROM STNAME
      EXEC SQL

```

```

OPEN C1
END-EXEC.
PERFORM PTSQLCA.
IF SQLCODE IS EQUAL TO ZERO
    PERFORM FETCH-DELETE-SITE-1
    PERFORM CLOSE-CURSOR-SITE-1.

*****
* Retrieve information about the transferring employee.      *
* Provided that the employee exists, perform DELETE-SITE-1 to *
* delete the employee from STLEC1.                          *
*****

FETCH-DELETE-SITE-1.

MOVE 'FETCH C1      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
END-EXEC.
PERFORM PTSQLCA.
IF SQLCODE IS EQUAL TO ZERO
    PERFORM DELETE-SITE-1.

*****
* Delete the employee from STLEC1.                          *
*****

DELETE-SITE-1.

MOVE 'DELETE EMPLOYEE ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
MOVE 'DELETE EMPLOYEE      ' TO STNAME
EXEC SQL
    DELETE FROM SYSADM.EMP
    WHERE EMPNO = :TEMP-EMPNO
END-EXEC.
PERFORM PTSQLCA.

*****
* Close the cursor used to retrieve information about the    *
* transferring employee.                                    *
*****

CLOSE-CURSOR-SITE-1.

MOVE 'CLOSE CURSOR C1      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    CLOSE C1
END-EXEC.
PERFORM PTSQLCA.

*****
* Update certain employee information in order to make it    *
* current.                                                    *
*****

UPDATE-ADDRESS.
MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
MOVE TEMP-CITY-LN         TO H-CITY-LN.
MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
MOVE 'SJCA'               TO H-EMPLOC.

*****
* Establish a connection to STLEC2                            *
*****

```

```

CONNECT-TO-SITE-2.

MOVE 'CONNECT TO STLEC2 ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    CONNECT TO :SITE-2
END-EXEC.
PERFORM PTSQLCA.

*****
* Using the employee information that was retrieved from STLEC1 *
* and updated previously, insert the employee at STLEC2.      *
*****

PROCESS-SITE-2.

MOVE 'INSERT EMPLOYEE ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    INSERT INTO SYSADM.EMP VALUES
        (:H-EMPNO,
         :H-NAME,
         :H-ADDRESS,
         :H-CITY,
         :H-EMPLOC,
         :H-SSNO,
         :H-BORN,
         :H-SEX,
         :H-HIRED,
         :H-DEPTNO,
         :H-JOBCODE,
         :H-SRATE,
         :H-EDUC,
         :H-SAL,
         :H-VALIDCHK)
END-EXEC.
PERFORM PTSQLCA.

*****
* COMMIT any changes that were made at STLEC1 and STLEC2.    *
*****

COMMIT-WORK.

MOVE 'COMMIT WORK ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    COMMIT
END-EXEC.
PERFORM PTSQLCA.

*****
* Include COBOL standard language procedures                  *
*****

INCLUDE-SUBS.
EXEC SQL INCLUDE COBSSUB END-EXEC.

```

Sample COBOL program using aliases for three-part names

You can access distributed data by using aliases for three-part names in a COBOL program.

The following sample program demonstrates distributed access data using aliases for three-part names with two-phase commit.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TWOPHASE.
AUTHOR.
REMARKS.
*****
*
* MODULE NAME = TWOPHASE
*
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING
* TWO PHASE COMMIT AND DRDA WITH
* ALIASES FOR THREE-PART NAMES
*
* FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS
* USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE
* FROM ONE LOCATION TO ANOTHER.
*
* NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE
* TABLE SYSADM.ALLEMPLOYEES AT LOCATIONS STLEC1
* AND STLEC2.
*
* MODULE TYPE = COBOL PROGRAM
* PROCESSOR = DB2 PRECOMPILER, ENTERPRISE COBOL FOR Z/OS
* MODULE SIZE = SEE LINK EDIT
* ATTRIBUTES = NOT REENTRANT OR REUSABLE
*
* ENTRY POINT =
* PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT
* LINKAGE = INVOKE FROM DSN RUN
* INPUT = NONE
* OUTPUT =
* SYMBOLIC LABEL/NAME = SYSPRINT
* DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH
* STEP AND THE RESULTANT SQLCA
*
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION
*
* EXIT ERROR = NONE
*
* EXTERNAL REFERENCES =
* ROUTINE SERVICES = NONE
* DATA-AREAS = NONE
* CONTROL-BLOCKS =
* SQLCA - SQL COMMUNICATION AREA
*
* TABLES = NONE
*
* CHANGE-ACTIVITY = NONE
*
*
* PSEUDOCODE
*
* MAINLINE.
* Perform PROCESS-CURSOR-SITE-1 to obtain the information
* about an employee that is transferring to another
* location.
* If the information about the employee was obtained
* successfully Then
* Do.
* | Perform UPDATE-ADDRESS to update the information to
* | contain current information about the employee.
* | Perform PROCESS-SITE-2 to insert the employee
* | information at the location where the employee is
* | transferring to.
* End if the employee information was obtained
* successfully.
* Perform COMMIT-WORK to COMMIT the changes made to STLEC1
*

```



```

*          and STLEC2.
*
*      PROG-END.
*      Close the printer.
*      Return.
*
*      PROCESS-CURSOR-SITE-1.
*      Provide a text description of the following step.
*      Open a cursor that will be used to retrieve information
*      about the transferring employee from this site.
*      Print the SQLCA out.
*      If the cursor was opened successfully Then
*      Do.
*          Perform FETCH-DELETE-SITE-1 to retrieve and
*          delete the information about the transferring
*          employee from this site.
*          Perform CLOSE-CURSOR-SITE-1 to close the cursor.
*      End if the cursor was opened successfully.
*
*      FETCH-DELETE-SITE-1.
*      Provide a text description of the following step.
*      Fetch information about the transferring employee.
*      Print the SQLCA out.
*      If the information was retrieved successfully Then
*      Do.
*          Perform DELETE-SITE-1 to delete the employee
*          at this site.
*      End if the information was retrieved successfully.
*
*      DELETE-SITE-1.
*      Provide a text description of the following step.
*      Delete the information about the transferring employee
*      from this site.
*      Print the SQLCA out.
*
*      CLOSE-CURSOR-SITE-1.
*      Provide a text description of the following step.
*      Close the cursor used to retrieve information about
*      the transferring employee.
*      Print the SQLCA out.
*
*      UPDATE-ADDRESS.
*      Update the address of the employee.
*      Update the city of the employee.
*      Update the location of the employee.
*
*      PROCESS-SITE-2.
*      Provide a text description of the following step.
*      Insert the employee information at the location where
*      the employee is being transferred to.
*      Print the SQLCA out.
*
*      COMMIT-WORK.
*      COMMIT all the changes made to STLEC1 and STLEC2.
*
*****

```

```

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PRINTER, ASSIGN TO S-OUT1.

```

```

DATA DIVISION.
FILE SECTION.
FD PRINTER
RECORD CONTAINS 120 CHARACTERS

```

```

DATA RECORD IS PRT-TC-RESULTS
LABEL RECORD IS OMITTED.
01 PRT-TC-RESULTS.
03 PRT-BLANK PIC X(120).

```

```

WORKING-STORAGE SECTION.

```

```

*****
* Variable declarations
*****

```

```

01 H-EMPTBL.
05 H-EMPNO PIC X(6).
05 H-NAME.
49 H-NAME-LN PIC S9(4) COMP-4.
49 H-NAME-DA PIC X(32).
05 H-ADDRESS.
49 H-ADDRESS-LN PIC S9(4) COMP-4.
49 H-ADDRESS-DA PIC X(36).
05 H-CITY.
49 H-CITY-LN PIC S9(4) COMP-4.
49 H-CITY-DA PIC X(36).
05 H-EMPLOC PIC X(4).
05 H-SSNO PIC X(11).
05 H-BORN PIC X(10).
05 H-SEX PIC X(1).
05 H-HIRED PIC X(10).
05 H-DEPTNO PIC X(3).
05 H-JOBCODE PIC S9(3)V COMP-3.
05 H-SRATE PIC S9(5) COMP.
05 H-EDUC PIC S9(5) COMP.
05 H-SAL PIC S9(6)V9(2) COMP-3.
05 H-VALIDCHK PIC S9(6)V COMP-3.
01 H-EMPTBL-IND-TABLE.
02 H-EMPTBL-IND PIC S9(4) COMP OCCURS 15 TIMES.

```

```

*****
* Includes for the variables used in the COBOL standard
* language procedures and the SQLCA.
*****

```

```

EXEC SQL INCLUDE COBSVAR END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.

```

```

*****
* Declaration for the table that contains employee information
*****

```

```

EXEC SQL DECLARE SYSADM.ALLEMPLOYEES TABLE
(EMPNO CHAR(6) NOT NULL,
NAME VARCHAR(32),
ADDRESS VARCHAR(36) ,
CITY VARCHAR(36) ,
EMPLOC CHAR(4) NOT NULL,
SSNO CHAR(11),
BORN DATE,
SEX CHAR(1),
HIRED CHAR(10),
DEPTNO CHAR(3) NOT NULL,
JOBCODE DECIMAL(3),
SRATE SMALLINT,
EDUC SMALLINT,
SAL DECIMAL(8,2) NOT NULL,
VALCHK DECIMAL(6))
END-EXEC.

```

```

*****

```

```

* Constants
*****

77 TEMP-EMPNO          PIC X(6)  VALUE '080000'.
77 TEMP-ADDRESS-LN    PIC 99    VALUE 15.
77 TEMP-CITY-LN       PIC 99    VALUE 18.

*****
* Declaration of the cursor that will be used to retrieve
* information about a transferring employee
* EC1EMP is the alias for STLEC1.SYSADM.ALLEMPLOYEES
*****

EXEC SQL DECLARE C1 CURSOR FOR
      SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
             SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
             SRATE, EDUC, SAL, VALCHK
      FROM   EC1EMP
      WHERE EMPNO = :TEMP-EMPNO
END-EXEC.
PROCEDURE DIVISION.
A101-HOUSE-KEEPING.
OPEN OUTPUT PRINTER.

*****
* An employee is transferring from location STLEC1 to STLEC2.
* Retrieve information about the employee from STLEC1, delete
* the employee from STLEC1 and insert the employee at STLEC2
* using the information obtained from STLEC1.
*****

MAINLINE.
PERFORM PROCESS-CURSOR-SITE-1
IF SQLCODE IS EQUAL TO 0
PERFORM UPDATE-ADDRESS
PERFORM PROCESS-SITE-2.
PERFORM COMMIT-WORK.

PROG-END.
CLOSE PRINTER.
GOBACK.

*****
* Open the cursor that will be used to retrieve information
* about the transferring employee.
*****

PROCESS-CURSOR-SITE-1.

MOVE 'OPEN CURSOR C1      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
OPEN C1
END-EXEC.
PERFORM PTSQLCA.
IF SQLCODE IS EQUAL TO ZERO
PERFORM FETCH-DELETE-SITE-1
PERFORM CLOSE-CURSOR-SITE-1.

*****
* Retrieve information about the transferring employee.
* Provided that the employee exists, perform DELETE-SITE-1 to
* delete the employee from STLEC1.
*****

FETCH-DELETE-SITE-1.

```

```

MOVE 'FETCH C1      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
END-EXEC.      PERFORM PTSQLCA.
IF SQLCODE IS EQUAL TO ZERO
    PERFORM DELETE-SITE-1.

```

```

*****
* Delete the employee from STLEC1.      *
*****

```

DELETE-SITE-1.

```

MOVE 'DELETE EMPLOYEE ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
MOVE 'DELETE EMPLOYEE      ' TO STNAME
EXEC SQL
    DELETE FROM EC1EMP
    WHERE EMPNO = :TEMP-EMPNO
END-EXEC.
PERFORM PTSQLCA.

```

```

*****
* Close the cursor used to retrieve information about the      *
* transferring employee.      *
*****

```

CLOSE-CURSOR-SITE-1.

```

MOVE 'CLOSE CURSOR C1      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    CLOSE C1
END-EXEC.
PERFORM PTSQLCA.

```

```

*****
* Update certain employee information in order to make it      *
* current.      *
*****

```

UPDATE-ADDRESS.

```

MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
MOVE TEMP-CITY-LN        TO H-CITY-LN.
MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
MOVE 'SJCA'              TO H-EMPLOC.

```

```

*****
* Using the employee information that was retrieved from STLEC1 *
* and updated previously, insert the employee at STLEC2.      *
* EC2EMP is the alias for STLEC2.SYSADM.ALLEMPLOYEES      *
*****

```

PROCESS-SITE-2.

```

MOVE 'INSERT EMPLOYEE      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
    INSERT INTO EC2EMP VALUES
    (:H-EMPNO,
    :H-NAME,
    :H-ADDRESS,
    :H-CITY,
    :H-EMPLOC,
    :H-SSNO,
    :H-BORN,

```

```

:H-SEX,
:H-HIRED,
:H-DEPTNO,
:H-JOBCODE,
:H-SRATE,
:H-EDUC,
:H-SAL,
:H-VALIDCHK)
END-EXEC.
PERFORM PTSQLCA.

```

```

*****
* COMMIT any changes that were made at STLEC1 and STLEC2.      *
*****

```

```

COMMIT-WORK.

```

```

MOVE 'COMMIT WORK      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
  COMMIT
END-EXEC.
PERFORM PTSQLCA.

```

```

*****
* Include COBOL standard language procedures                    *
*****

```

```

INCLUDE-SUBS.
EXEC SQL INCLUDE COBSSUB END-EXEC.

```

Example COBOL stored procedure with a GENERAL WITH NULLS linkage convention

You can call a stored procedure that uses the GENERAL WITH NULLS linkage convention from a COBOL program.

This example stored procedure does the following:

- Searches the DB2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```

CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE COBOL
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO

```

```

RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 2
COMMIT ON RETURN NO;

```

The following example is a COBOL stored procedure with linkage convention GENERAL WITH NULLS.

```

CBL RENT
IDENTIFICATION DIVISION.
PROGRAM-ID. GETPRML.
AUTHOR. EXAMPLE.
DATE-WRITTEN. 03/25/98.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
*
WORKING-STORAGE SECTION.
*
EXEC SQL INCLUDE SQLCA END-EXEC.
*
*****
* DECLARE A HOST VARIABLE TO HOLD INPUT SCHEMA
*****
01 INSCHEMA PIC X(8).
*****
* DECLARE CURSOR FOR RETURNING RESULT SETS
*****
*
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INSCHEMA
END-EXEC.
*
LINKAGE SECTION.
*****
* DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
*****
01 PROCNM PIC X(18).
01 SCHEMA PIC X(8).
*****
* DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
*****
01 OUT-CODE PIC S9(9) USAGE BINARY.
01 PARMLST.
49 PARMLST-LEN PIC S9(4) USAGE BINARY.
49 PARMLST-TEXT PIC X(254).
*****
* DECLARE THE STRUCTURE CONTAINING THE NULL
* INDICATORS FOR THE INPUT AND OUTPUT PARAMETERS.
*****
01 IND-PARM.
03 PROCNM-IND PIC S9(4) USAGE BINARY.
03 SCHEMA-IND PIC S9(4) USAGE BINARY.
03 OUT-CODE-IND PIC S9(4) USAGE BINARY.
03 PARMLST-IND PIC S9(4) USAGE BINARY.

PROCEDURE DIVISION USING PROCNM, SCHEMA,
OUT-CODE, PARMLST, IND-PARM.
*****
* If any input parameter is null, return a null value
* for PARMLST and set the output return code to 9999.
*****
IF PROCNM-IND < 0 OR

```

```

        SCHEMA-IND < 0
        MOVE 9999 TO OUT-CODE
        MOVE 0 TO OUT-CODE-IND
        MOVE -1 TO PARMLST-IND
    ELSE
*****
* Issue the SQL SELECT against the SYSIBM.SYSROUTINES
* DB2 catalog table.
*****
    EXEC SQL
        SELECT RUNOPTS INTO :PARMLST
        FROM SYSIBM.SYSROUTINES
        WHERE NAME=:PROCNM AND
        SCHEMA=:SCHEMA
    END-EXEC
        MOVE 0 TO PARMLST-IND
*****
* COPY SQLCODE INTO THE OUTPUT PARAMETER AREA
*****
        MOVE SQLCODE TO OUT-CODE
        MOVE 0 TO OUT-CODE-IND.
*
*****
* OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
* TO THE CALLER.
*****
    EXEC SQL OPEN C1
    END-EXEC.
    PROG-END.
    GOBACK.

```

Example COBOL stored procedure with a GENERAL linkage convention

You can call a stored procedure that uses the GENERAL linkage convention from a COBOL program.

This example stored procedure does the following:

- Searches the catalog table SYSROUTINES for a row matching the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

This stored procedure is able to return a NULL value for the output host variables.

The linkage convention for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```

CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
    OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
    LANGUAGE COBOL
    DETERMINISTIC
    READS SQL DATA
    EXTERNAL NAME "GETPRML"
    COLLID GETPRML
    ASUTIME NO LIMIT

```

```

PARAMETER STYLE GENERAL
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 2
COMMIT ON RETURN NO;

CBL RENT
IDENTIFICATION DIVISION.
PROGRAM-ID. GETPRML.
AUTHOR. EXAMPLE.
DATE-WRITTEN. 03/25/98.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

      EXEC SQL INCLUDE SQLCA END-EXEC.
*****
*   DECLARE A HOST VARIABLE TO HOLD INPUT SCHEMA
*****
01  INSCHEMA PIC X(8).

*****
*   DECLARE CURSOR FOR RETURNING RESULT SETS
*****
*
      EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
          SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INSCHEMA
          END-EXEC.
*
LINKAGE SECTION.
*****
*   DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
*****
01  PROCNM  PIC X(18).
01  SCHEMA  PIC X(8).
*****
*   DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
*****
01  OUT-CODE PIC S9(9) USAGE BINARY.
01  PARMLST.
    49 PARMLST-LEN PIC S9(4) USAGE BINARY.
    49 PARMLST-TEXT PIC X(254).

PROCEDURE DIVISION USING PROCNM, SCHEMA,
    OUT-CODE, PARMLST.

*****
* Issue the SQL SELECT against the SYSIBM.SYSROUTINES
* DB2 catalog table.
*****
      EXEC SQL
          SELECT RUNOPTS INTO :PARMLST
          FROM SYSIBM.ROUTINES
          WHERE NAME=:PROCNM AND
          SCHEMA=:SCHEMA
          END-EXEC.

*****
* COPY SQLCODE INTO THE OUTPUT PARAMETER AREA
*****

```



```

        MOVE SQLCODE TO OUT-CODE.
*****
* OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
* TO THE CALLER.
*****
        EXEC SQL OPEN C1
        END-EXEC.
    PROG-END.
        GOBACK.

```

Example COBOL program that calls a stored procedure

You can call the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention from a COBOL program on a z/OS system.

Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets. The following figure contains the example COBOL program that calls the GETPRML stored procedure.

```

IDENTIFICATION DIVISION.
PROGRAM-ID.    CALPRML.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT REPOUT
        ASSIGN TO UT-S-SYSPRINT.

DATA DIVISION.
FILE SECTION.
FD  REPOUT
    RECORD CONTAINS 127 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS REPREC.
01  REPREC          PIC X(127).

WORKING-STORAGE SECTION.
*****
* MESSAGES FOR SQL CALL
*****
01  SQLREC.
    02  BADMSG      PIC X(34) VALUE
        ' SQL CALL FAILED DUE TO SQLCODE = '.
    02  BADCODE     PIC +9(5) USAGE DISPLAY.
    02  FILLER      PIC X(80) VALUE SPACES.
01  ERRMREC.
    02  ERRMSG      PIC X(12) VALUE ' SQLERRMC = '.
    02  ERRMCODE    PIC X(70).
    02  FILLER      PIC X(38) VALUE SPACES.
01  CALLREC.
    02  CALLMSG     PIC X(28) VALUE
        ' GETPRML FAILED DUE TO RC = '.
    02  CALLCODE    PIC +9(5) USAGE DISPLAY.
    02  FILLER      PIC X(42) VALUE SPACES.
01  RSLTREC.
    02  RSLTMSG     PIC X(15) VALUE
        ' TABLE NAME IS '.
    02  TBLNAME     PIC X(18) VALUE SPACES.
    02  FILLER      PIC X(87) VALUE SPACES.

*****
* WORK AREAS
*****
01  PROCNM          PIC X(18).
01  SCHEMA          PIC X(8).
01  OUT-CODE        PIC S9(9) USAGE COMP.
01  PARMLST.

```

```

    49 PARMLN          PIC S9(4) USAGE COMP.
    49 PARMTXT        PIC X(254).
01  PARMBUF REDEFINES PARMLST.
    49 PARBLN          PIC S9(4) USAGE COMP.
    49 PARMARRY       PIC X(127) OCCURS 2 TIMES.
01  NAME.
    49 NAMELEN        PIC S9(4) USAGE COMP.
    49 NAMETXT        PIC X(18).
    77 PARMIND        PIC S9(4) COMP.
    77 I              PIC S9(4) COMP.
    77 NUMLINES       PIC S9(4) COMP.
*****
* DECLARE A RESULT SET LOCATOR FOR THE RESULT SET *
* THAT IS RETURNED. *
*****
01  LOC              USAGE SQL TYPE IS
                        RESULT-SET-LOCATOR VARYING.

*****
* SQL INCLUDE FOR SQLCA *
*****
      EXEC SQL INCLUDE SQLCA  END-EXEC.

PROCEDURE DIVISION.
*-----
PROG-START.
      OPEN OUTPUT REPOUT.
*          OPEN OUTPUT FILE
      MOVE 'DSN8EP2      ' TO PROCNM.
*          INPUT PARAMETER -- PROCEDURE TO BE FOUND
      MOVE SPACES TO SCHEMA.
*          INPUT PARAMETER -- SCHEMA IN SYSROUTINES
      MOVE -1 TO PARMIND.
*          THE PARMLST PARAMETER IS AN OUTPUT PARM.
*          MARK PARMLST PARAMETER AS NULL, SO THE DB2
*          REQUESTER DOES NOT HAVE TO SEND THE ENTIRE
*          PARMLST VARIABLE TO THE SERVER. THIS
*          HELPS REDUCE NETWORK I/O TIME, BECAUSE
*          PARMLST IS FAIRLY LARGE.
      EXEC SQL
        CALL GETPRML(:PROCNM,
                    :SCHEMA,
                    :OUT-CODE,
                    :PARMLST INDICATOR :PARMIND)
      END-EXEC.

*          MAKE THE CALL
      IF SQLCODE NOT EQUAL TO +466 THEN
*          IF CALL RETURNED BAD SQLCODE
          MOVE SQLCODE TO BADCODE
          WRITE REPREC FROM SQLREC
          MOVE SQLERRMC TO ERRMCODE
          WRITE REPREC FROM ERRMREC
        ELSE
          PERFORM GET-PARMS
          PERFORM GET-RESULT-SET.
PROG-END.
      CLOSE REPOUT.
*          CLOSE OUTPUT FILE
      GOBACK.
PARMPRT.
      MOVE SPACES TO REPREC.
      WRITE REPREC FROM PARMARRY(I)
        AFTER ADVANCING 1 LINE.
GET-PARMS.
*          IF THE CALL WORKED,
      IF OUT-CODE NOT EQUAL TO 0 THEN
*          DID GETPRML HIT AN ERROR?

```

```

        MOVE OUT-CODE TO CALLCODE
        WRITE REPREC FROM CALLREC
    ELSE
*         EVERYTHING WORKED
        DIVIDE 127 INTO PARMLN GIVING NUMLINES ROUNDED
*         FIND OUT HOW MANY LINES TO PRINT
        PERFORM PARMPT VARYING I
            FROM 1 BY 1 UNTIL I GREATER THAN NUMLINES.
    GET-RESULT-SET.
*****
* ASSUME YOU KNOW THAT ONE RESULT SET IS RETURNED, *
* AND YOU KNOW THE FORMAT OF THAT RESULT SET. *
* ALLOCATE A CURSOR FOR THE RESULT SET, AND FETCH *
* THE CONTENTS OF THE RESULT SET. *
*****
        EXEC SQL ASSOCIATE LOCATORS (:LOC)
            WITH PROCEDURE GETPRML
        END-EXEC.
*         LINK THE RESULT SET TO THE LOCATOR
    EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :LOC
    END-EXEC.
*         LINK THE CURSOR TO THE RESULT SET
    PERFORM GET-ROWS VARYING I
        FROM 1 BY 1 UNTIL SQLCODE EQUAL TO +100.
    GET-ROWS.
        EXEC SQL FETCH C1 INTO :NAME
        END-EXEC.
        MOVE NAME TO TBLNAME.
        WRITE REPREC FROM RSLTREC
            AFTER ADVANCING 1 LINE.

```

Chapter 7. Coding SQL statements in Fortran application programs

When you code SQL statements in Fortran application programs, you should follow certain guidelines.

Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran

Fortran programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

To define the SQL communications area, SQLSTATE, and SQLCODE:

Choose one of the following actions:

Option	Description
To define the SQL communications area:	<ol style="list-style-type: none">1. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration: <pre>EXEC SQL INCLUDE SQLCA</pre>DB2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.

Option	Description
<p>To declare SQLCODE and SQLSTATE host variables:</p>	<ol style="list-style-type: none"> <li data-bbox="938 233 1409 436">1. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as INTEGER*4. This variable can also be called SQLCOD. <li data-bbox="938 447 1409 651">2. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as CHARACTER*5. This variable can also be called SQLCOD. <p data-bbox="938 678 1409 730">Restriction: Do not declare an SQLSTATE variable as an element of a structure.</p> <p data-bbox="938 737 1409 863">Requirement: After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</p>

Related tasks:

“Checking the execution of SQL statements” on page 227

“Checking the execution of SQL statements by using the SQLCA” on page 228

“Checking the execution of SQL statements by using SQLCODE and SQLSTATE” on page 232

“Defining the items that your program can use to check whether an SQL statement executed successfully” on page 173

Defining SQL descriptor areas in Fortran

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

To define SQL descriptor areas:

Call a subroutine that is written in C, PL/I, or assembler language and that uses the INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions that you need.

Restrictions:

- You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.
- You cannot use the SQL INCLUDE statement for the SQLDA, because it is not supported in COBOL.

Related tasks:

“Defining SQL descriptor areas” on page 173

Declaring host variables and indicator variables in Fortran

You can use host variables, host variable arrays, and host structures in SQL statements in your program to pass data between DB2 and your application.

To declare host variables, host variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:
 - When you declare a character host variable, do not use an expression to define the length of the character variable. You can use a character host variable with an undefined length (for example, CHARACTER *(*)). The length of any such variable is determined when the associated SQL statement executes.
 - Host variables must be scalar variables; they cannot be elements of vectors or arrays (subscripted variables).
 - Be careful when calling subroutines that might change the attributes of a host variable. Such alteration can cause an error while the program is running.
 - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
 - If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
 - Ensure that any SQL statement that uses a host variable or host variable array is within the scope of the statement that declares that variable or array.
 - If you are using the DB2 precompiler, ensure that the names of host variables and host variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.
2. Optional: Define any associated indicator variables, arrays, and structures.

Related tasks:

“Declaring host variables and indicator variables” on page 174

Host variables in Fortran

In Fortran programs, you can specify numeric, character, LOB, and ROWID host variables. You can also specify result set and LOB locators.

Restrictions:

- Only some of the valid Fortran declarations are valid host variable declarations. If the declaration for a variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- Fortran supports some data types with no SQL equivalent (for example, REAL*16 and COMPLEX). In most cases, you can use Fortran statements to convert between the unsupported data types and the data types that SQL allows.
- You can not use locators as column types.

The following locator data types are Fortran data types and SQL data types:

- Result set locator

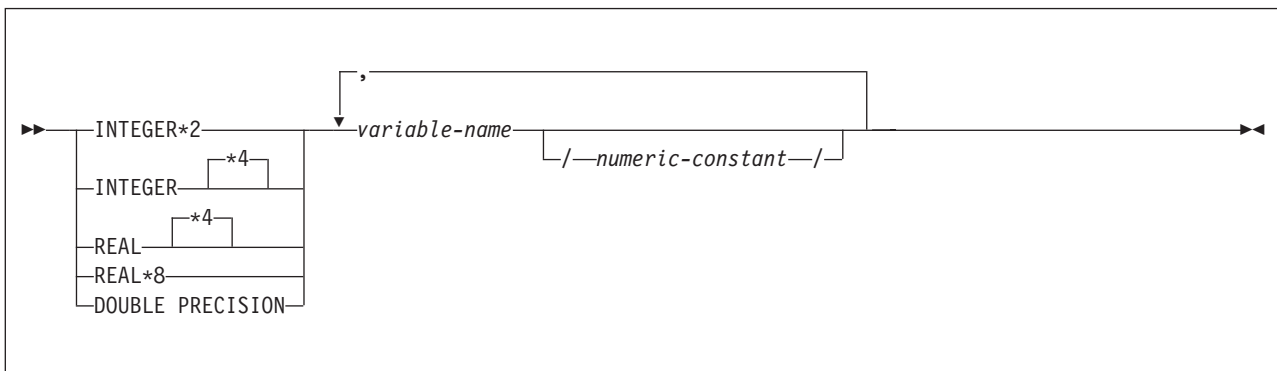
- LOB locators
- Because Fortran does not support graphic data types, Fortran applications can process only Unicode tables that use UTF-8 encoding.

Recommendations:

- Be careful of overflow. For example, if you retrieve an INTEGER column value into a INTEGER*2 host variable and the column value is larger than 32767 or -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.
- Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHARACTER*70 host variable, the rightmost ten characters of the retrieved string are truncated. Retrieving a double-precision floating-point or decimal column value into an INTEGER*4 host variable removes any fractional value.

Numeric host variables

The following diagram shows the syntax for declaring numeric host variables.

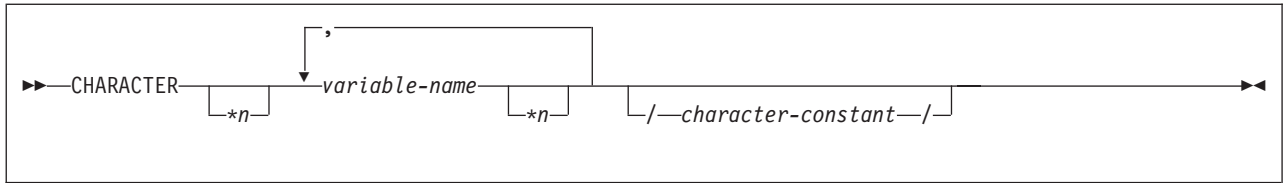


Restrictions:

- Fortran does not provide an equivalent for the decimal data type. To hold a decimal value, use one of the following variables:
 - An integer or floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, use a floating-point variable. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.
 - A character string host variable. Use the CHAR function to retrieve a decimal value into it.
- The SQL data type DECFLOAT has no equivalent in Fortran.

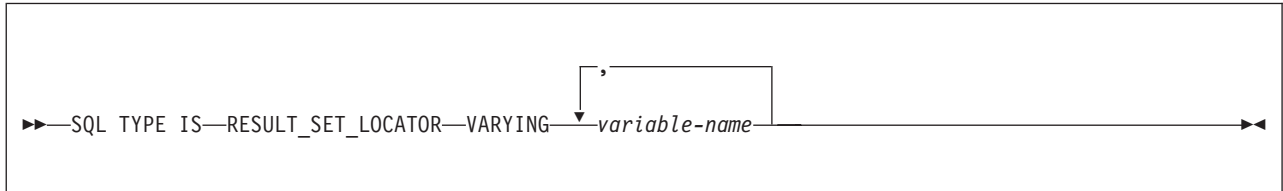
Character host variables

The following diagram shows the syntax for declaring character host variables other than CLOBs.



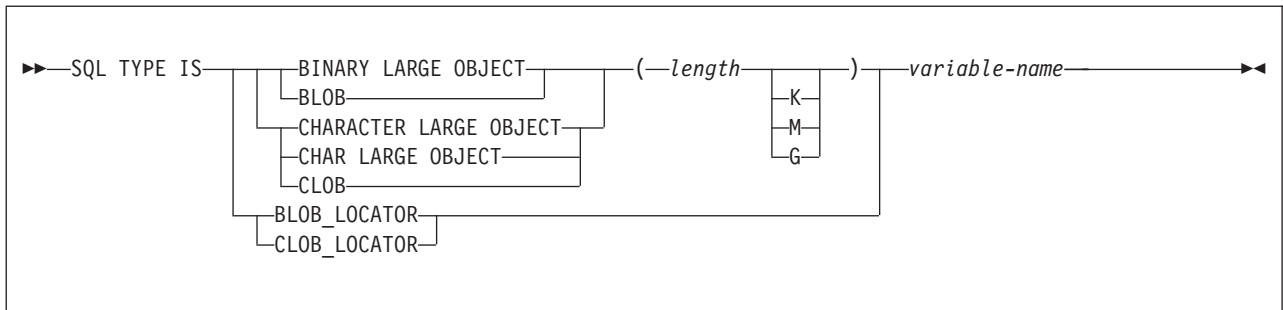
Result set locators

The following diagram shows the syntax for declaring result set locators.



LOB variables and locators

The following diagram shows the syntax for declaring BLOB and CLOB host variables and locators.



ROWID host variables

The following diagram shows the syntax for declarations of ROWID variables.



Constants

The syntax for constants in Fortran programs differs from the syntax for constants in SQL statements in the following ways:

- Fortran interprets a string of digits with a decimal point to be a real constant. An SQL statement interprets such a string to be a decimal constant. Therefore, use exponent notation when specifying a real (that is, floating-point) constant in an SQL statement.

- In Fortran, a real (floating-point) constant that has a length of 8 bytes uses a D as the exponent indicator (for example, 3.14159D+04). An 8-byte floating-point constant in an SQL statement must use an E (for example, 3.14159E+04).

Related concepts:

“Host variables” on page 174

“Rules for host variables in an SQL statement” on page 183

“Large objects (LOBs)” on page 465

Related tasks:

“Determining whether a retrieved value in a host variable is null or truncated” on page 186

“Inserting a single row by using a host variable” on page 189

“Inserting null values into columns by using indicator variables or arrays” on page 190

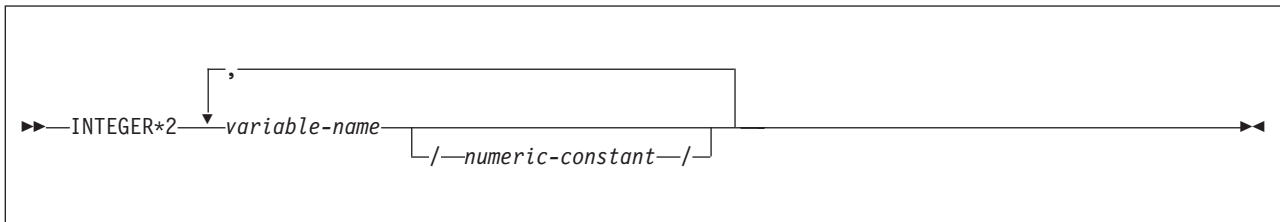
“Retrieving a single row of data into host variables” on page 184

“Updating data by using host variables” on page 189

Indicator variables in Fortran

An indicator variable is a 2-byte integer (INTEGER*2). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in Fortran.



Example

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
C                               :DAY :DAYIND,
C                               :BGN :BGNIND,
C                               :END :ENDIND
```

You can declare these variables as follows:

```
CHARACTER*7 CLSCD
INTEGER*2 DAY
CHARACTER*8 BGN, END
INTEGER*2 DAYIND, BGNIND, ENDIND
```

Related concepts:

“Indicator variables, arrays, and structures” on page 176

Related tasks:

“Inserting null values into columns by using indicator variables or arrays” on page 190

Equivalent SQL and Fortran data types

When you declare host variables in your Fortran programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

Table 68. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in Fortran programs

Fortran host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
INTEGER*2	500	2	SMALLINT
INTEGER*4	496	4	INTEGER
REAL*4	480	4	FLOAT (single precision)
REAL*8	480	8	FLOAT (double precision)
CHARACTER*n	452	n	CHAR(n)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator. Do not use this data type as a column type.
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator. Do not use this data type as a column type.
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator. Do not use this data type as a column type.
SQL TYPE IS BLOB(n) 1≤n≤2147483647	404	n	BLOB(n)
SQL TYPE IS CLOB(n) 1≤n≤2147483647	408	n	CLOB(n)
SQL TYPE IS ROWID	904	40	ROWID

Notes:

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.

The following table shows equivalent Fortran host variables for each SQL data type. Use this table to determine the Fortran data type for host variables that you define to receive output from the database. For example, if you retrieve `TIMESTAMP` data, you can define a variable of type `CHARACTER*n`.

This table shows direct conversions between SQL data types and Fortran data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 converts those compatible data types.

Table 69. Fortran host variable equivalents that you can use when retrieving data of a particular SQL data type

SQL data type	Fortran host variable equivalent	Notes
SMALLINT	INTEGER*2	
INTEGER	INTEGER*4	
BIGINT	not supported	

Table 69. Fortran host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	Fortran host variable equivalent	Notes
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	no exact equivalent	Use REAL*8
FLOAT(<i>n</i>) single precision	REAL*4	1<= <i>n</i> <=21
FLOAT(<i>n</i>) double precision	REAL*8	22<= <i>n</i> <=53
CHAR(<i>n</i>)	CHARACTER* <i>n</i>	1<= <i>n</i> <=255
VARCHAR(<i>n</i>)	no exact equivalent	Use a character host variable that is large enough to contain the largest expected VARCHAR value.
BINARY	not supported	
VARBINARY	not supported	
GRAPHIC(<i>n</i>)	not supported	
VARGRAPHIC(<i>n</i>)	not supported	
DATE	CHARACTER* <i>n</i>	If you are using a date exit routine, <i>n</i> is determined by that routine; otherwise, <i>n</i> must be at least 10.
TIME	CHARACTER* <i>n</i>	If you are using a time exit routine, <i>n</i> is determined by that routine. Otherwise, <i>n</i> must be at least 6; to include seconds, <i>n</i> must be at least 8.
TIMESTAMP	CHARACTER* <i>n</i>	<i>n</i> must be at least 19. To include microseconds, <i>n</i> must be 26; if <i>n</i> is less than 26, truncation occurs on the microseconds part.
TIMESTAMP(0)	CHARACTER* <i>n</i>	<i>n</i> must be at least 19.
TIMESTAMP(<i>p</i>) <i>p</i> > 0	CHARACTER* <i>n</i>	<i>n</i> must be at least 19. To include fractional seconds, <i>n</i> must be 20+ <i>x</i> where <i>x</i> is the number of fractional seconds to include; if <i>x</i> is less than <i>p</i> , truncation occurs on the fractional seconds part.
TIMESTAMP(<i>p</i>) WITH TIME ZONE	no exact equivalent	Use a character host variable that is large enough to contain the largest expected timestamp with time zone value.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type. ¹
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type. ¹
DBCLOB locator	not supported	
BLOB(<i>n</i>)	SQL TYPE IS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647 ¹
CLOB(<i>n</i>)	SQL TYPE IS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647 ¹
DBCLOB(<i>n</i>)	not supported	
ROWID	SQL TYPE IS ROWID	

Table 69. Fortran host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	Fortran host variable equivalent	Notes
XML	not supported	

Related concepts:

“Compatibility of SQL and language data types” on page 180

“LOB host variable, LOB locator, and LOB file reference variable declarations” on page 757

SQL statements in Fortran programs

You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

Fortran source statements must be fixed-length 80-byte records. The DB2 precompiler does not support free-form source input.

Each SQL statement in a Fortran program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code the UPDATE statement in a Fortran program as follows:

```
EXEC SQL
C  UPDATE DSN8A10.DEPT
C  SET MGRNO = :MGRNUM
C  WHERE DEPTNO = :INTDEPT
```

You cannot follow an SQL statement with another SQL statement or Fortran statement on the same line.

Fortran does not require blanks to delimit words within a statement, but the SQL language requires blanks. The rules for embedded SQL follow the rules for SQL syntax, which require you to use one or more blanks as a delimiter.

Comments: You can include Fortran comment lines within embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can include SQL comments in any embedded SQL statement.

The DB2 precompiler does not support the exclamation point (!) as a comment recognition character in Fortran programs.

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for Fortran statements, except that you must specify EXEC SQL on one line. The SQL examples in this topic have Cs in the sixth column to indicate that they are continuations of EXEC SQL.

Declaring tables and views: Your Fortran program should also include the DECLARE TABLE statement to describe each table and view the program accesses.

Dynamic SQL in a Fortran program: In general, Fortran programs can easily handle dynamic SQL statements. SELECT statements can be handled if the data

types and the number of returned fields are fixed. If you want to use variable-list SELECT statements, you need to use an SQLDA, as described in “Defining SQL descriptor areas” on page 173.

You can use a Fortran character variable in the statements PREPARE and EXECUTE IMMEDIATE, even if it is fixed-length.

Including code: To include SQL statements or Fortran host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements. You cannot use the Fortran INCLUDE compiler directive to include SQL statements or Fortran host variable declarations.

Margins: Code the SQL statements between columns 7 through 72, inclusive. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid Fortran name for a host variable. Do not use external entry names that begin with 'DSN' or host variable names that begin with 'SQL'. These names are reserved for DB2.

Do not use the word DEBUG, except when defining a Fortran DEBUG packet. Do not use the words FUNCTION, IMPLICIT, PROGRAM, and SUBROUTINE to define variables.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement labels: You can specify statement numbers for SQL statements in columns 1 to 5. However, during program preparation, a labeled SQL statement generates a Fortran CONTINUE statement with that label before it generates the code that executes the SQL statement. Therefore, a labeled SQL statement should never be the last statement in a DO loop. In addition, you should not label SQL statements (such as INCLUDE and BEGIN DECLARE SECTION) that occur before the first executable SQL statement, because an error might occur.

WHENEVER statement: The target for the GOTO clause in the SQL WHENEVER statement must be a label in the Fortran source code and must refer to a statement in the same subprogram. The WHENEVER statement only applies to SQL statements in the same subprogram.

Special Fortran considerations: The following considerations apply to programs written in Fortran:

- You cannot use the @PROCESS statement in your source code. Instead, specify the compiler options in the PARM field.
- You cannot use the SQL INCLUDE statement to include the following statements: PROGRAM, SUBROUTINE, BLOCK, FUNCTION, or IMPLICIT.

DB2 supports Version 3 Release 1 (or later) of VS Fortran with the following restrictions:

- The parallel option is not supported. Applications that contain SQL statements must not use Fortran parallelism.

- You cannot use the byte data type within embedded SQL, because byte is not a recognizable host data type.

Handling SQL error return codes in Fortran:

You can use the subroutine DSNTIR to convert an SQL return code into a text message. DSNTIR builds a parameter list and calls DSNTIAR for you. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 229.

You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234.

DSNTIR syntax:

```
CALL DSNTIR ( error-length, message, return-code )
```

The DSNTIR parameters have the following meanings:

error-length

The total length of the message output area.

message

An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text are put into this area. For example, you could specify the format of the output area as:

```
INTEGER   ERRLEN /1320/
CHARACTER*132 ERRTXT(10)
INTEGER   ICODE
:
CALL DSNTIR ( ERRLEN, ERRTXT, ICODE )
```

where ERRLEN is the total length of the message output area, ERRTXT is the name of the message output area, and ICODE is the return code.

return-code

Accepts a return code from DSNTIAR.


An example of calling DSNTIR (which then calls DSNTIAR) from an application appears in the DB2 sample assembler program DSN8BF3, which is contained in the library DSN8A10.SDSNSAMP. See “DB2 sample applications” on page 1092 for instructions on how to access and print the source code for the sample program.

Related tasks:

“Including dynamic SQL in your program” on page 193

“Embedding SQL statements in your application” on page 183

“Handling SQL error codes” on page 239

 Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Delimiters in SQL statements in Fortran programs

You must delimit SQL statements in your Fortran program so that DB2 knows when a particular SQL statement ends.

Delimit an SQL statement in your Fortran program with the beginning keyword EXEC SQL and an end of line or end of last continued line.

Related reference:

“Programming examples” on page 251

Chapter 8. Coding SQL statements in PL/I application programs

When you code SQL statements in PL/I application programs, you should follow certain guidelines.

Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I

PL/I programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

To define the SQL communications area, SQLSTATE, and SQLCODE:

Choose one of the following actions:

Option	Description
To define the SQL communications area:	<ol style="list-style-type: none">1. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration: <pre>EXEC SQL INCLUDE SQLCA</pre> DB2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.

Option	Description
<p>To declare <code>SQLCODE</code> and <code>SQLSTATE</code> host variables:</p>	<ol style="list-style-type: none"> 1. Declare the <code>SQLCODE</code> variable within a <code>BEGIN DECLARE SECTION</code> statement and an <code>END DECLARE SECTION</code> statement in your program declarations as <code>BIN FIXED (31)</code>. 2. Declare the <code>SQLSTATE</code> variable within a <code>BEGIN DECLARE SECTION</code> statement and an <code>END DECLARE SECTION</code> statement in your program declarations as <code>CHARACTER(5)</code>. <p>Restriction: Do not declare an <code>SQLSTATE</code> variable as an element of a structure.</p> <p>Requirement: After you declare the <code>SQLCODE</code> and <code>SQLSTATE</code> variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</p>

Related tasks:

“Checking the execution of SQL statements” on page 227

“Checking the execution of SQL statements by using the `SQLCA`” on page 228

“Checking the execution of SQL statements by using `SQLCODE` and `SQLSTATE`” on page 232

“Defining the items that your program can use to check whether an SQL statement executed successfully” on page 173

Defining SQL descriptor areas in PL/I

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

To define SQL descriptor areas:

Code the SQLDA directly in the program, or use the following SQL INCLUDE statement to request a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA
```

Restriction: You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the `TWOPASS` SQL processing option.

Related tasks:

“Defining SQL descriptor areas” on page 173

Declaring host variables and indicator variables in PL/I

You can use host variables, host variable arrays, and host structures in SQL statements in your program to pass data between DB2 and your application.

To declare host variables, host variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:

- If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
- If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
- Ensure that any SQL statement that uses a host variable or host variable array is within the scope of the statement that declares that variable or array.
- If you are using the DB2 precompiler, ensure that the names of host variables and host variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.

2. Optional: Define any associated indicator variables, arrays, and structures.

Related tasks:

“Declaring host variables and indicator variables” on page 174

Host variables in PL/I

In PL/I programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

Restrictions:

- Only some of the valid PL/I declarations are valid host variable declarations. The precompiler uses the data attribute defaults that are specified in the PL/I DEFAULT statement. If the declaration for a host variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- The alignment, scope, and storage attributes of host variables have the following restrictions:
 - A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
 - If you use the BASED storage attribute, you must follow it with a PL/I element-locator-expression.
 - Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC storage class, or options. However, CICS requires that programs be reentrant.

Although the precompiler uses only the names and data attributes of variables and ignores the alignment, scope, and storage attributes, you should not ignore these restrictions. If you do ignore them, you might have problems compiling the PL/I source code that the precompiler generates.

- PL/I supports some data types with no SQL equivalent (COMPLEX and BIT variables, for example). In most cases, you can use PL/I statements to convert between the unsupported PL/I data types and the data types that SQL supports.
- You can not use locators as column types.

The following locator data types are PL/I data types as well as SQL data types:

- Result set locator
- Table locator

- LOB locators
- The precompiler does not support PL/I scoping rules.

Recommendations:

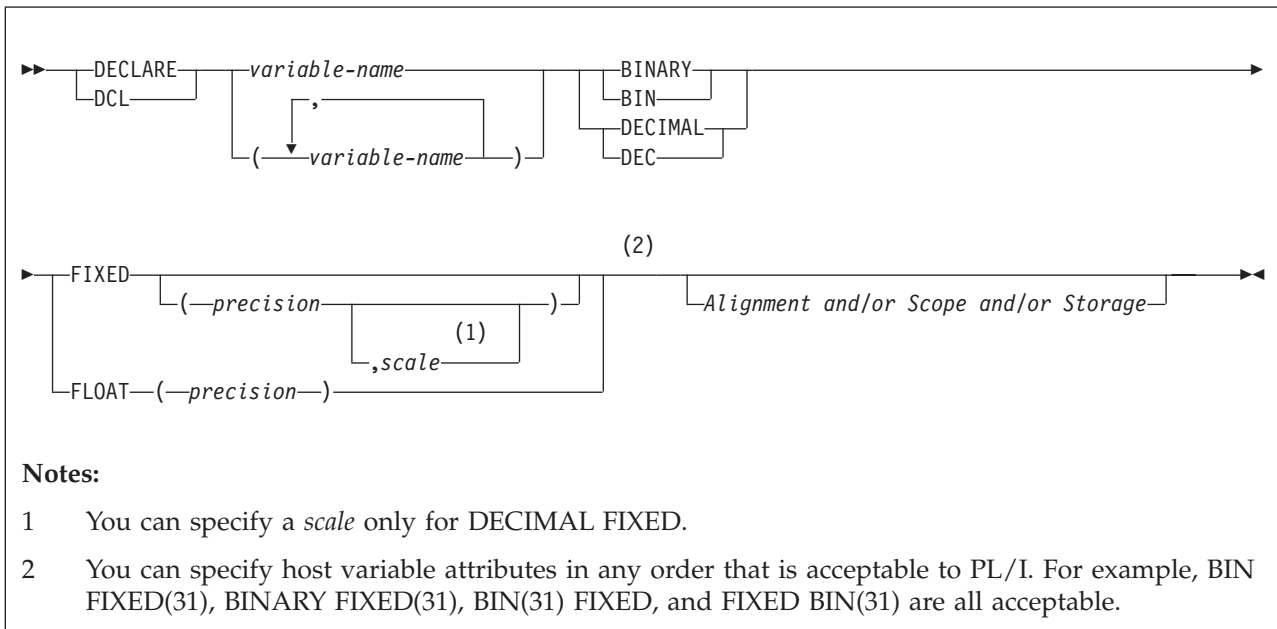
- Be careful of overflow. For example, if you retrieve an INTEGER column value into a BIN FIXED(15) host variable and the column value is larger than 32767 or smaller than -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.
- Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHAR(70) host variable, the rightmost ten characters of the retrieved string are truncated. Retrieving a double-precision floating-point or decimal column value into a BIN FIXED(31) host variable removes any fractional part of the value. Similarly, retrieving a column value with a DECIMAL data type into a PL/I decimal variable with a lower precision might truncate the value.

Numeric host variables

You can specify the following forms of numeric host variables:

- Floating-point numbers (Hexadecimal and Decimal)
- Integers and small integers
- Decimal numbers

The following diagram shows the syntax for declaring numeric host variables.



| For binary floating-point or hexadecimal floating-point data types, use the FLOAT
 | SQL processing option to specify whether the host variable is in IEEE binary
 | floating-point or z/Architecture hexadecimal floating-point format. DB2 does not
 | check if the format of the host variable contents match the format that you
 | specified with the FLOAT SQL processing option. Therefore, you need to ensure
 | that your floating-point host variable contents match the format that you specified
 | with the FLOAT SQL processing option. DB2 converts all floating-point input data
 | to z/Architecture hexadecimal floating-point format before storing it.

If the PL/I compiler that you are using does not support a decimal data type with a precision greater than 15, use one of the following variable types for decimal data:

- Decimal variables with precision less than or equal to 15, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, the fractional part of the value might truncate.
- An integer or a floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, use a floating-point variable. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

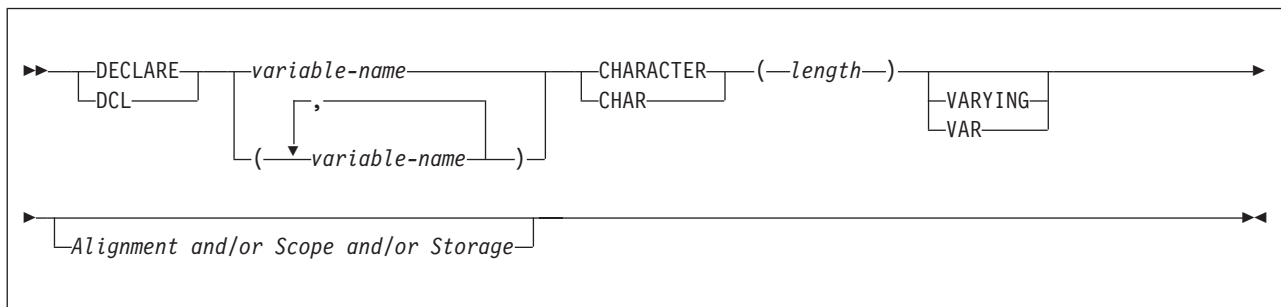
To use the PL/I decimal floating-point host data types, you need to use the FLOAT(DFP) and ARCH(7) compiler options and the DB2 coprocessor. The maximum precision for extended DECIMAL FLOAT will be 34 (not 33 as it is for hexadecimal float). The maximum precision for short DECIMAL FLOAT will be 7 (not 6 as it is for hexadecimal float).

Character host variables

You can specify the following forms of character host variables:

- Fixed-length strings
- Varying-length strings
- CLOBs

The following diagram shows the syntax for declaring character host variables, other than CLOBs.

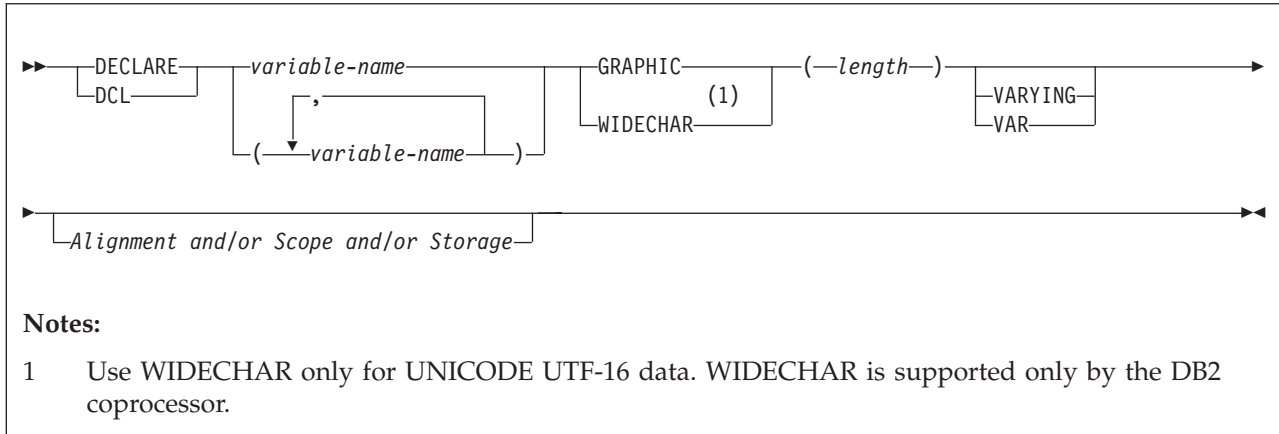


Graphic host variables

You can specify the following forms of character host variables:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following diagram shows the syntax for declaring graphic host variables other than DBCLOBs.

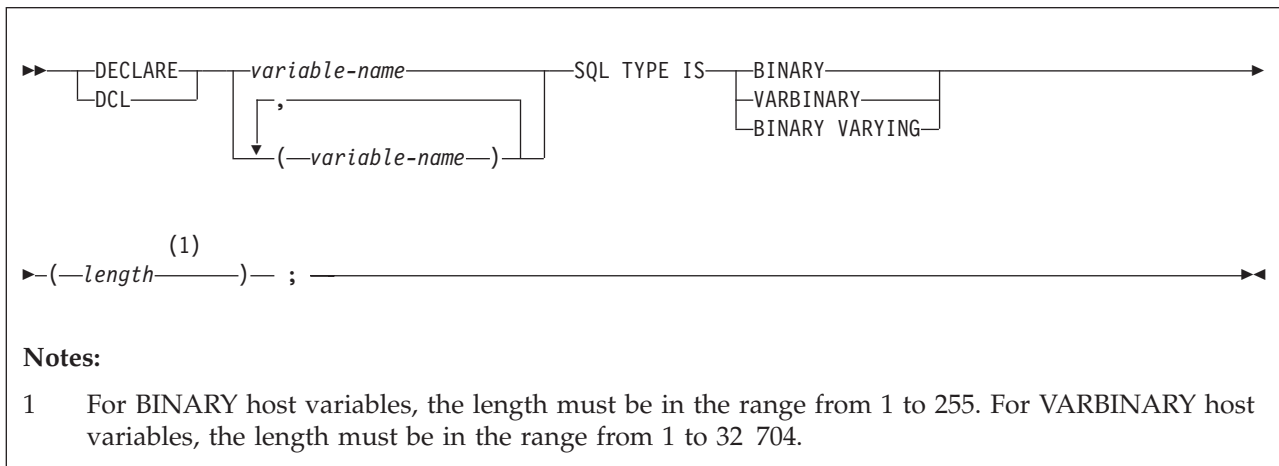


Binary host variables

You can specify the following forms of binary host variables:

- Fixed-length strings
- Varying-length strings
- BLOBs

The following diagram shows the syntax for declaring BINARY host variables.



PL/I does not have variables that correspond to the SQL binary data types BINARY and VARBINARY. To create host variables that can be used with these data types, use the SQL TYPE IS clause.

When you reference a BINARY or VARBINARY host variable in an SQL statement, you must use the variable that you specify in the SQL TYPE declaration. When you reference the host variable in a host language statement, you must use the variable that DB2 generates.

Examples of binary variable declarations: The following table shows examples of variables that DB2 generates when you declare binary host variables.

Table 70. Examples of BINARY and VARBINARY variable declarations for PL/I

Variable declaration that you include in your PL/I program	Corresponding variable that DB2 generates in the output source member
DCL BIN_VAR SQL TYPE IS BINARY(10);	DCL BIN_VAR CHAR(10);
DCL VBIN_VAR SQL TYPE IS VARBINARY(10);	DCL VBIN_VAR CHAR(10) VAR;

Result set locators

The following diagram shows the syntax for declaring result set locators.

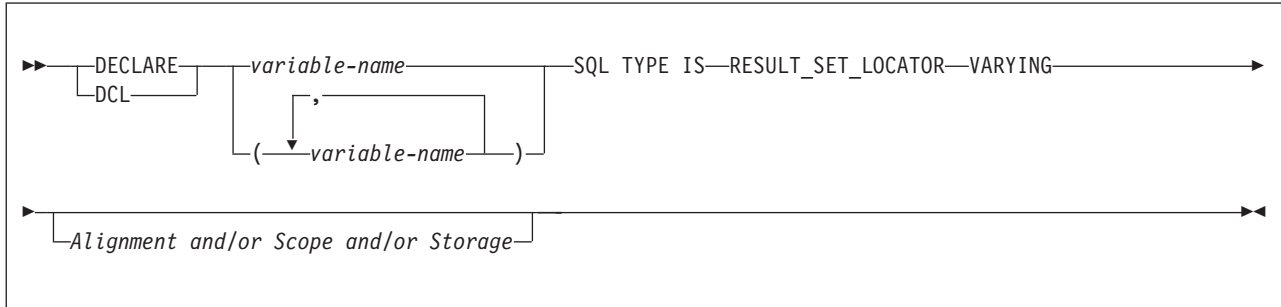
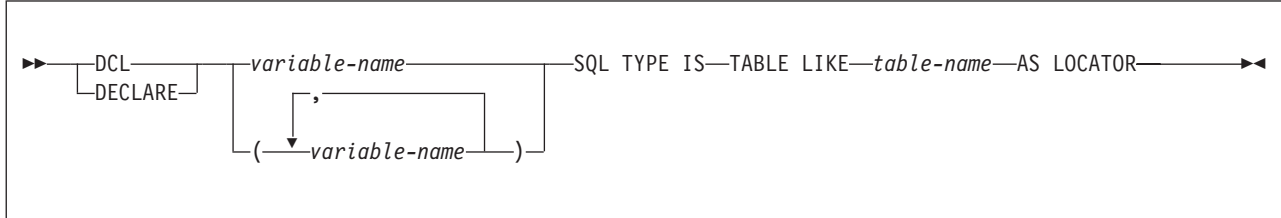


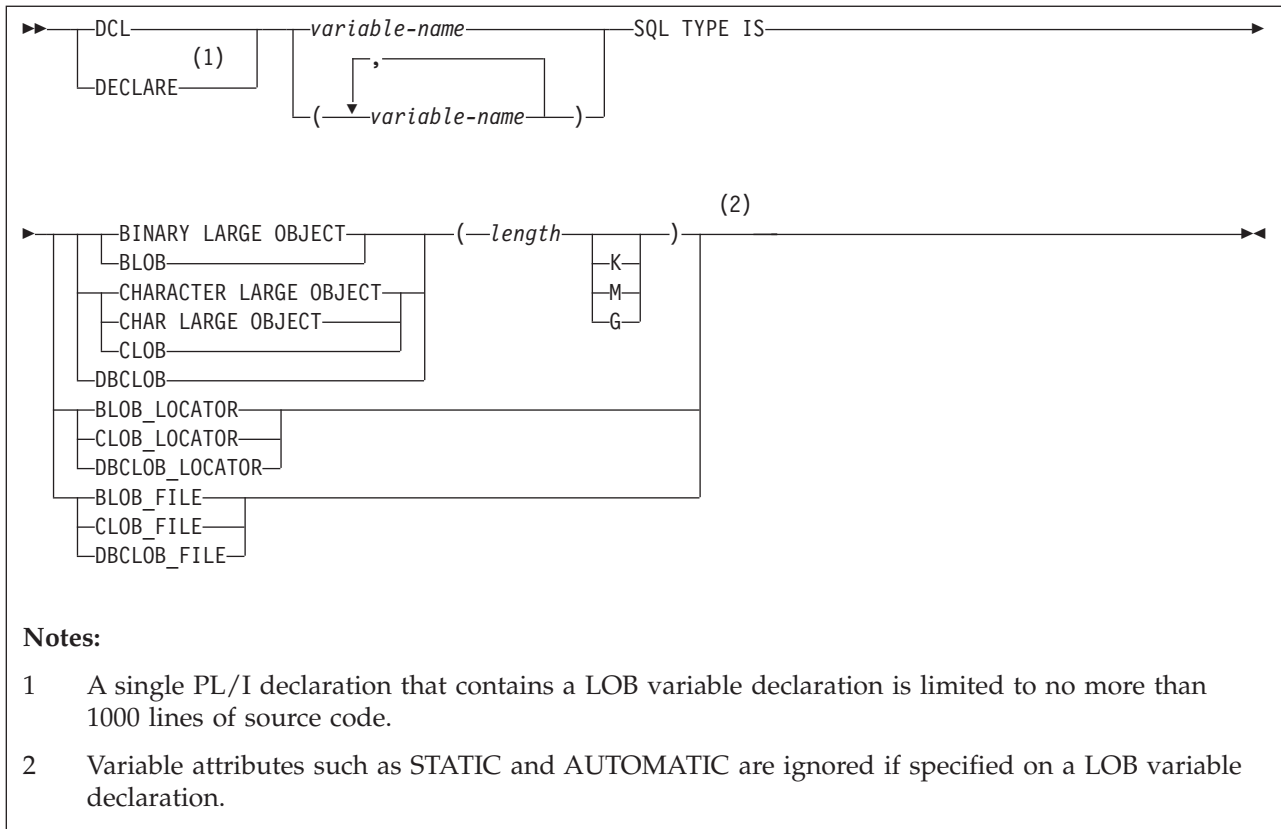
Table locators

The following diagram shows the syntax for declaring table locators.



LOB variables, locators, and file reference variables

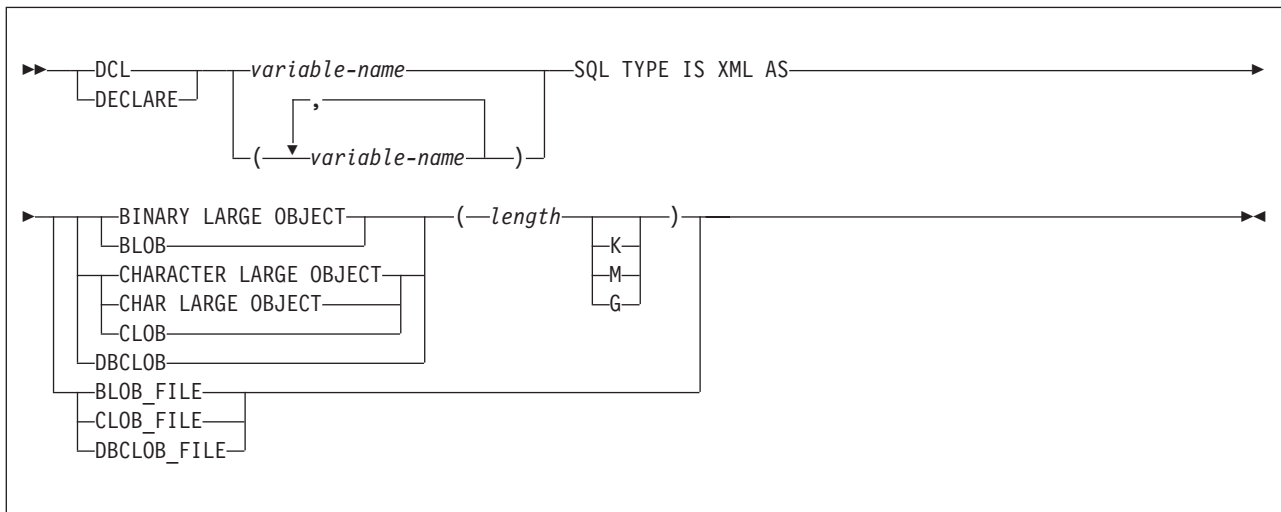
The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables, locators, and file reference variables.



Note: Variable attributes such as STATIC and AUTOMATIC are ignored if specified on a LOB variable declaration.

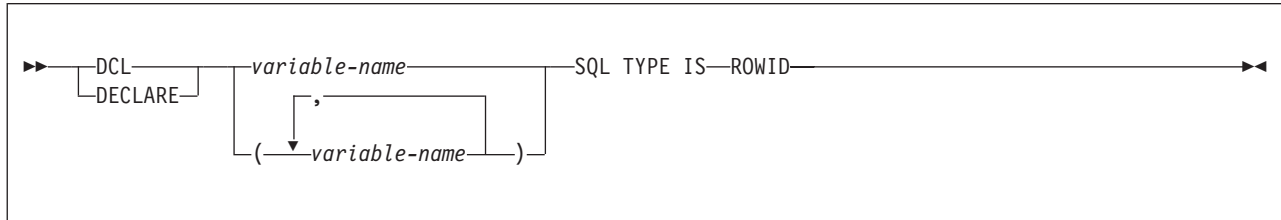
XML data host and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.



ROWID host variables

The following diagram shows the syntax for declaring ROWID host variables.



Related concepts:

“Host variables” on page 174

“Rules for host variables in an SQL statement” on page 183

“Large objects (LOBs)” on page 465

 Decimal floating-point (DECFLOAT) (DB2 SQL)

Related tasks:

“Determining whether a retrieved value in a host variable is null or truncated” on page 186

“Inserting a single row by using a host variable” on page 189

“Inserting null values into columns by using indicator variables or arrays” on page 190

“Retrieving a single row of data into host variables” on page 184

“Retrieving a single row of data into a host structure” on page 193

“Updating data by using host variables” on page 189

Host variable arrays in PL/I

In PL/I programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Restrictions:

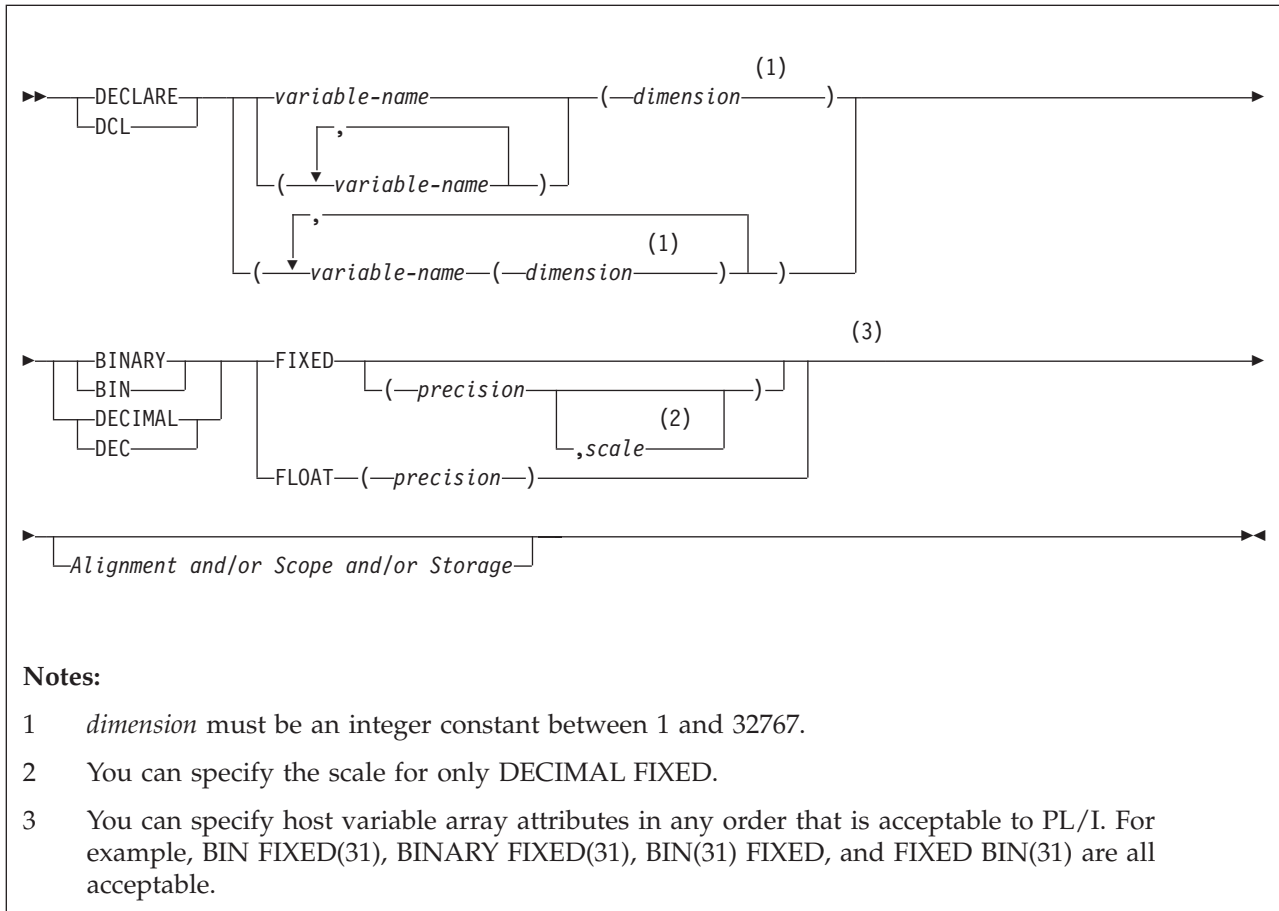
- Only some of the valid PL/I declarations are valid host variable declarations. The precompiler uses the data attribute defaults that are specified in the PL/I DEFAULT statement. If the declaration for a host variable is not valid, any SQL statement that references the host variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.
- The alignment, scope, and storage attributes of host variable arrays have the following restrictions:
 - A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
 - If you use the BASED storage attribute, you must follow it with a PL/I element-locator-expression.
 - Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC storage class, or options. However, CICS requires that programs be reentrant.

Although the precompiler uses only the names and data attributes of variable arrays and ignores the alignment, scope, and storage attributes, you should not ignore these restrictions. If you do ignore them, you might have problems compiling the PL/I source code that the precompiler generates.

- You must specify the ALIGNED attribute when you declare varying-length character arrays or varying-length graphic arrays that are to be used in multiple-row INSERT and FETCH statements.

Numeric host variable arrays

The following diagram shows the syntax for declaring numeric host variable arrays.

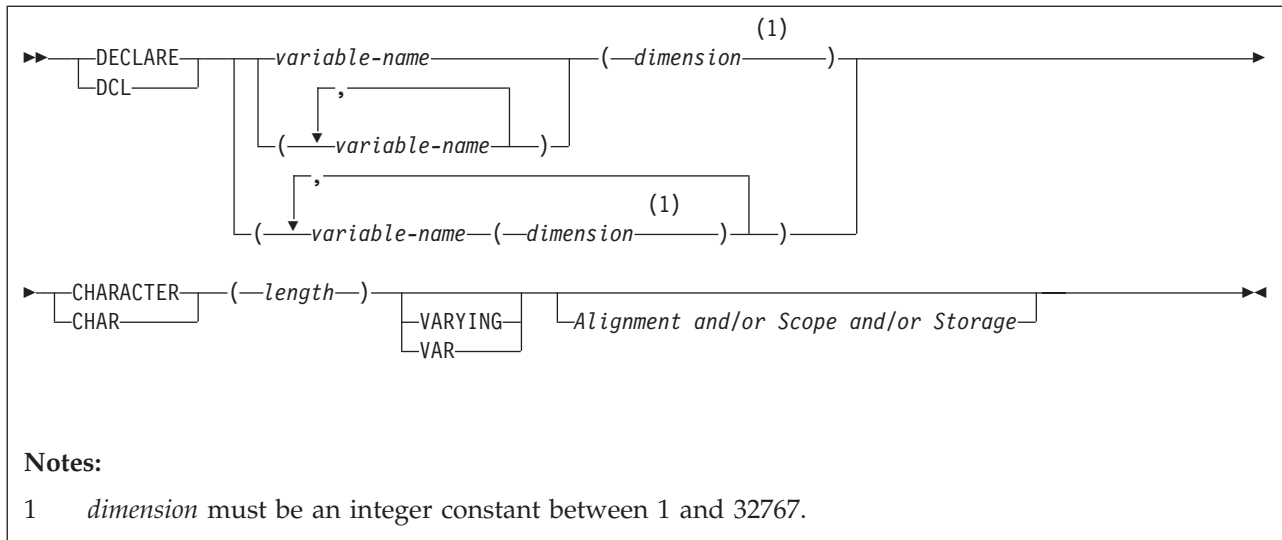


Example: The following example shows a declaration of an indicator array.
`DCL IND_ARRAY(100) BIN FIXED(15); /* DCL ARRAY of 100 indicator variables */`

To use the PL/I decimal floating-point host data types, you need to use the `FLOAT(DFP)` and `ARCH(7)` compiler options and the DB2 coprocessor. The maximum precision for extended `DECIMAL FLOAT` will be 34 (not 33 as it is for hexadecimal float). The maximum precision for short `DECIMAL FLOAT` will be 7 (not 6 as it is for hexadecimal float).

Character host variable arrays

The following diagram shows the syntax for declaring character host variable arrays other than CLOBs.

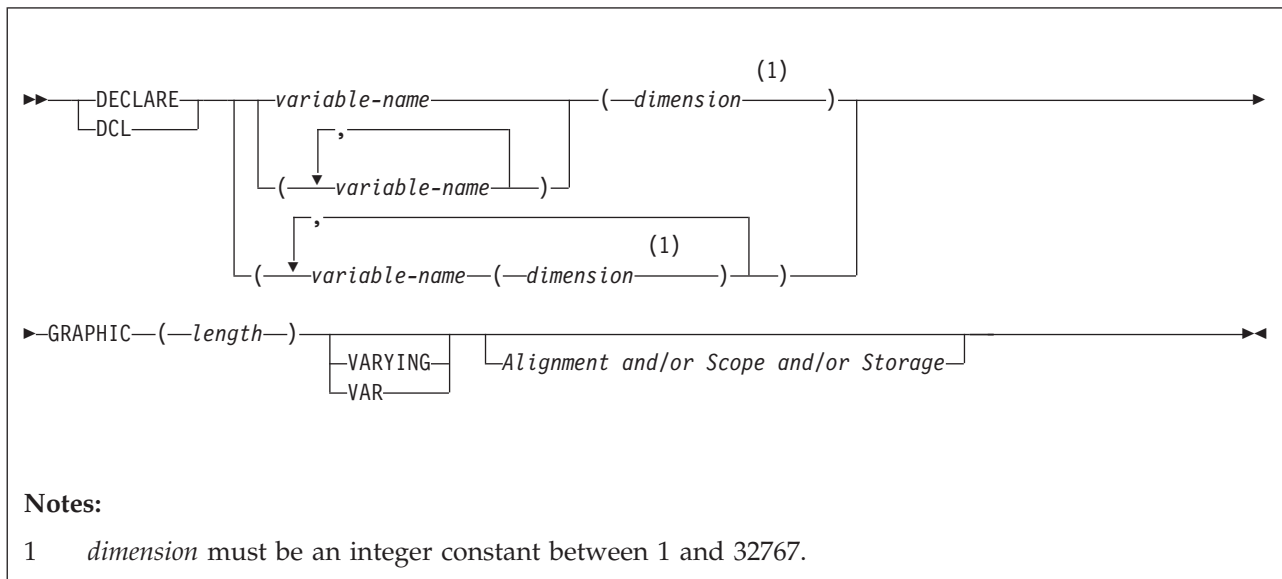


Example: The following example shows the declarations needed to retrieve 10 rows of the department number and name from the department table:

```
DCL DEPTNO(10)    CHAR(3);           /* Array of ten CHAR(3) variables */
DCL DEPTNAME(10) CHAR(29) VAR;      /* Array of ten VARCHAR(29) variables */
```

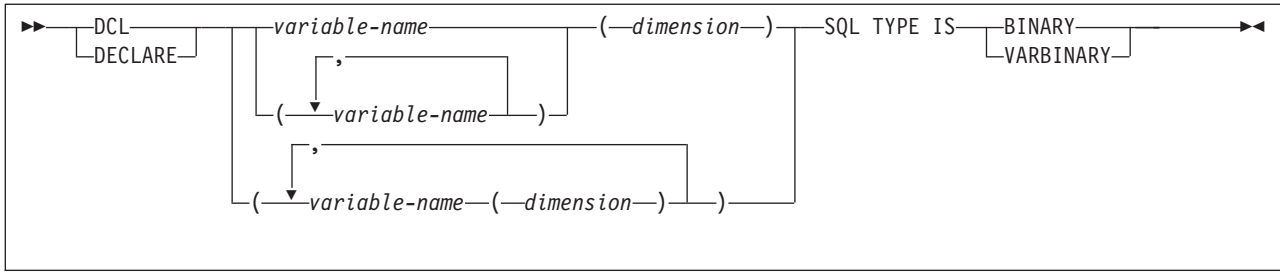
Graphic host variable arrays

The following diagram shows the syntax for declaring graphic host variable arrays other than DBCLOBs.



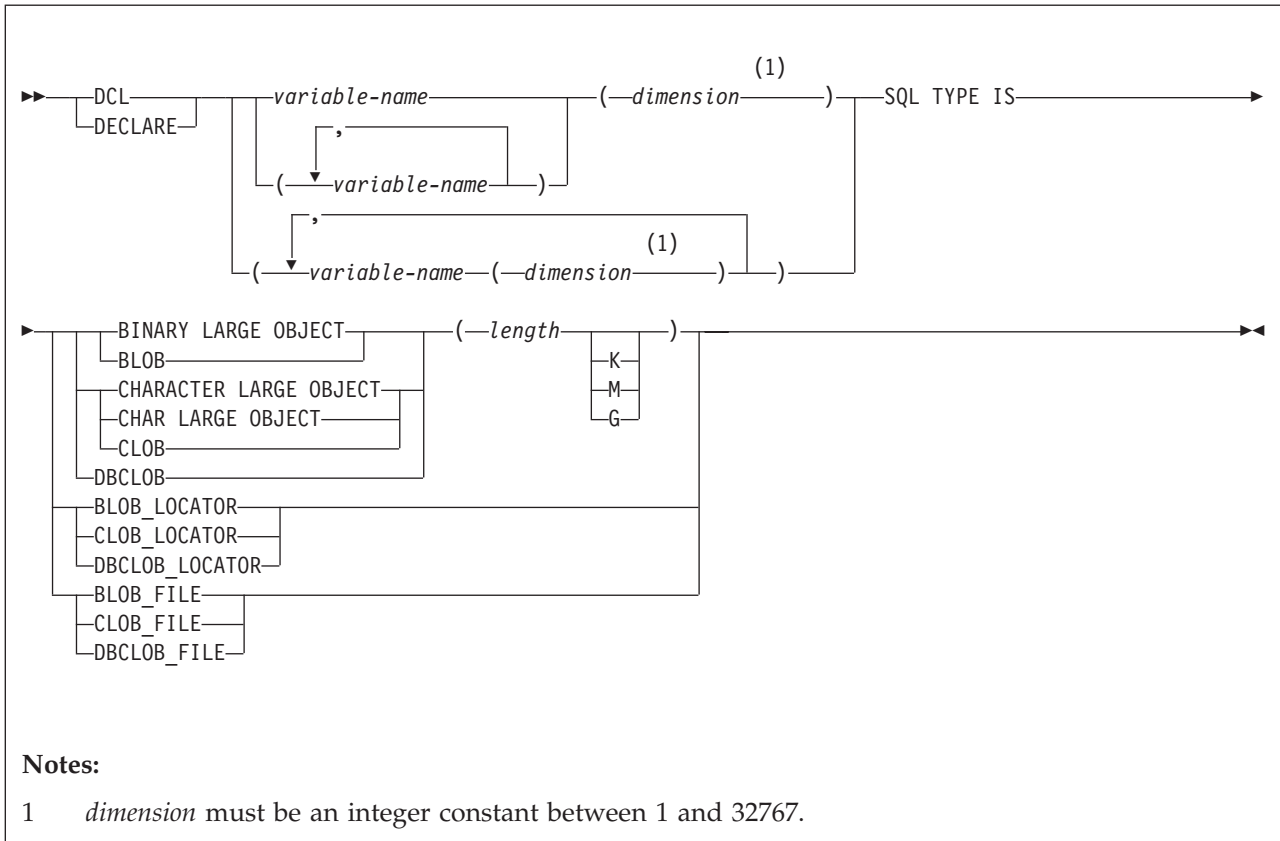
Binary host variable arrays

The following diagram shows the syntax for declaring binary variable arrays.



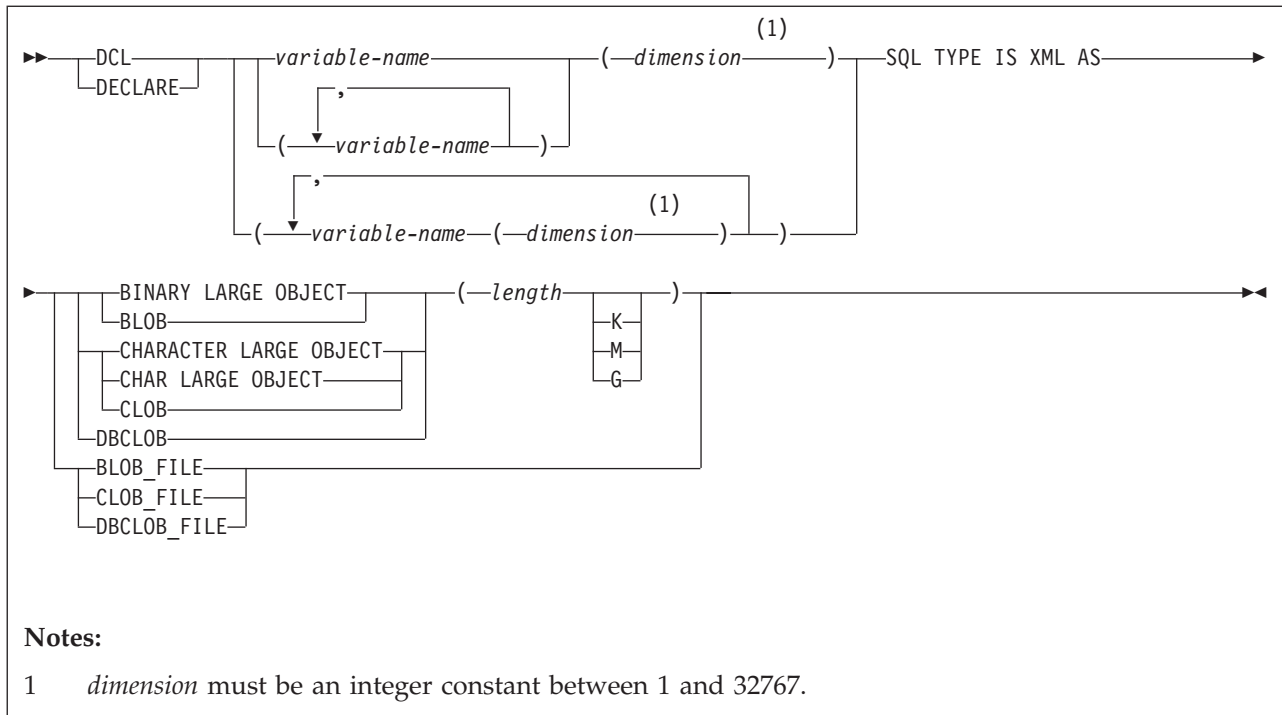
LOB, locator, and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable, locator, and file reference variable arrays.



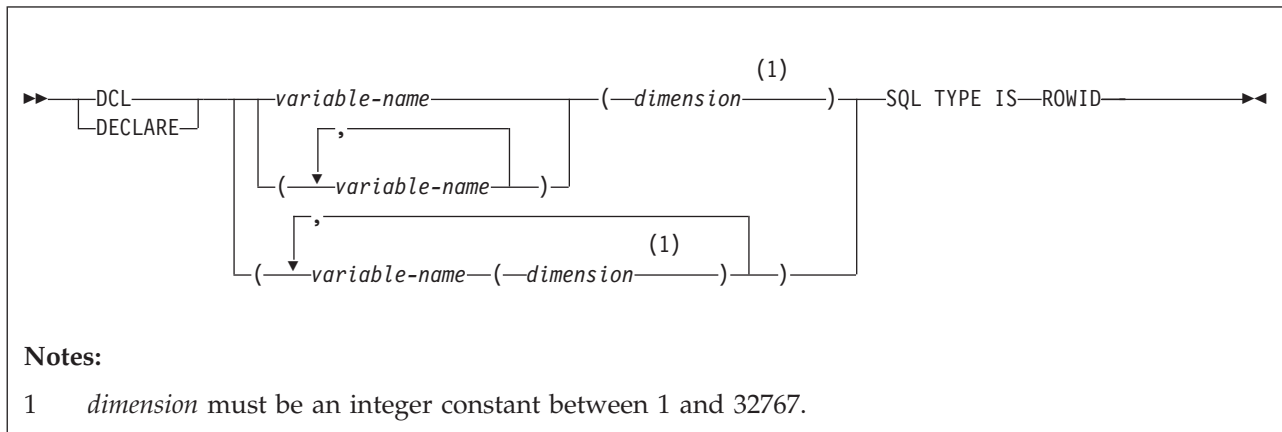
XML host and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable arrays and file reference variable arrays for XML data types.



ROWID variable arrays

The following diagram shows the syntax for declaring ROWID variable arrays.



Related concepts:

“Host variable arrays in an SQL statement” on page 191

“Host variable arrays” on page 175

“Large objects (LOBs)” on page 465

 Decimal floating-point (DECFLOAT) (DB2 SQL)

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

“Retrieving multiple rows of data into host variable arrays” on page 192

Host structures in PL/I

A PL/I host structure is a structure that contains subordinate levels of scalars. You can use the name of the structure as shorthand notation to reference the list of scalars.

Requirements: Host structure declarations in PL/I must satisfy the following requirements:

- Host structures are limited to two levels.
- You must terminate the host structure variable by ending the declaration with a semicolon.

Example:

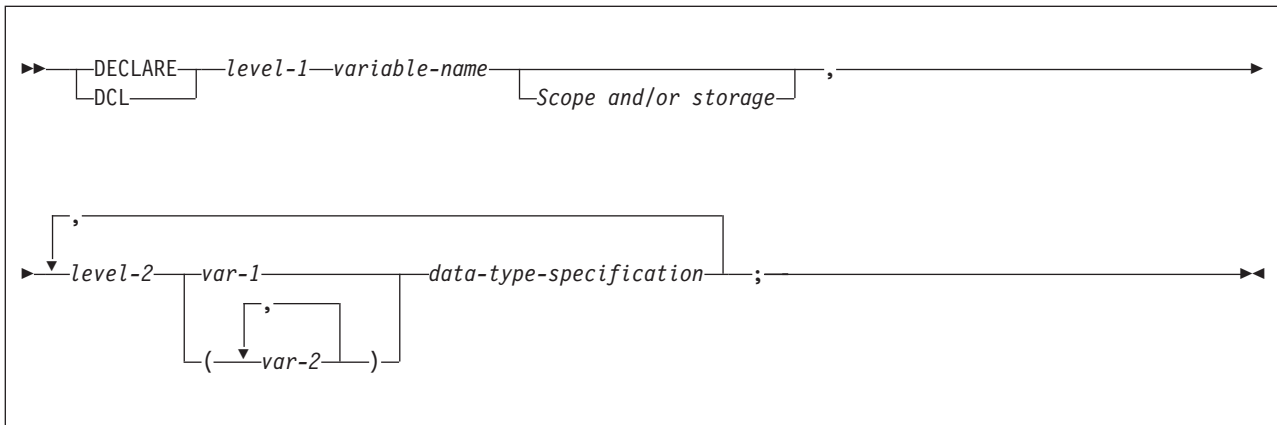
```
DCL 1 A,  
    2 B CHAR,  
    2 (C, D) CHAR;  
DCL (E, F) CHAR;
```

- You can specify host variable attributes in any order that is acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

When you reference a host variable, you can qualify it with a structure name. For example, you can specify STRUCTURE.FIELD.

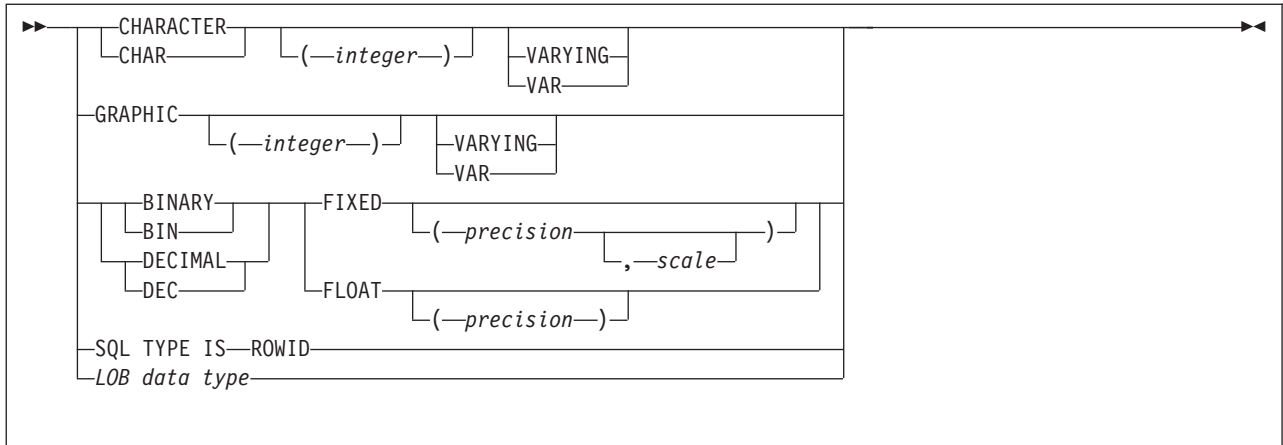
Host structures

The following diagram shows the syntax for declaring host structures.



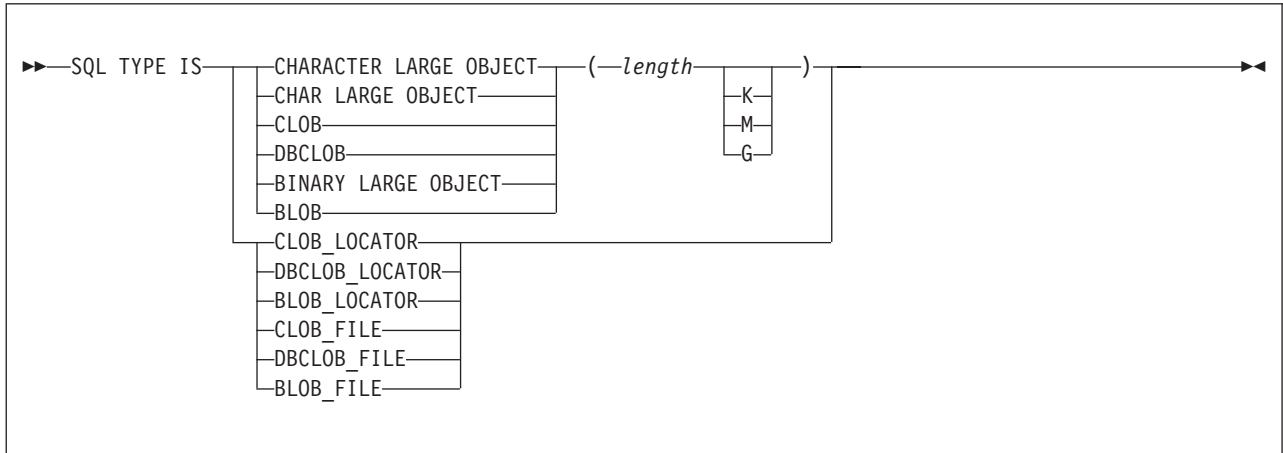
Data types

The following diagram shows the syntax for data types that are used within declarations of host structures.



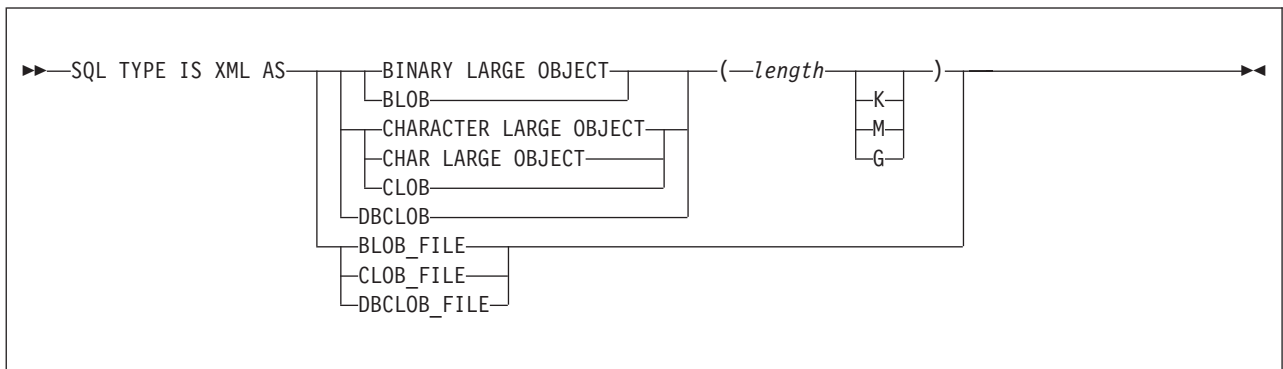
LOB data types

The following diagram shows the syntax for LOB data types that are used within declarations of host structures.



LOB data types for XML data

The following diagram shows the syntax for LOB data types that are used within declarations of host structures for XML data.



Example

In the following example, B is the name of a host structure that contains the scalars C1 and C2.

```
DCL 1 A,  
    2 B,  
    3 C1 CHAR(...),  
    3 C2 CHAR(...);
```

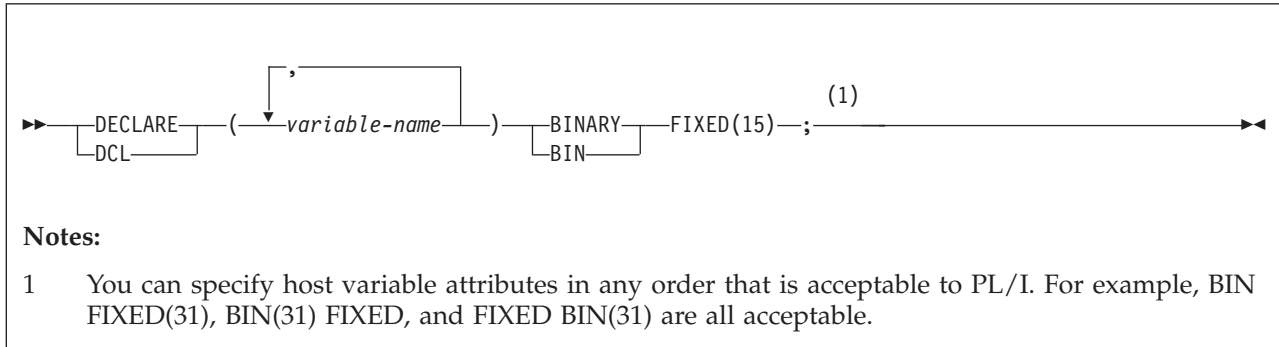
Related concepts:

“Host structures” on page 175

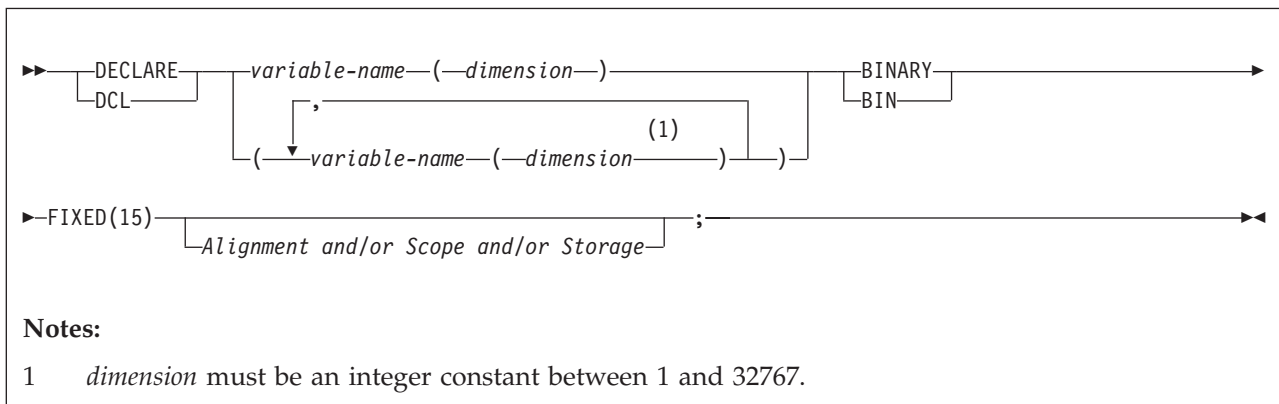
Indicator variables in PL/I

An indicator variable is a 2-byte integer (or an integer declared as BIN FIXED(15)). An indicator variable array is an array of 2-byte integers. You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in PL/I.



The following diagram shows the syntax for declaring an indicator array in PL/I.



Example

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.


```
EXEC SQL FETCH CLS_CURSOR INTO :CLS_CD,
                                :DAY :DAY_IND,
                                :BGN :BGN_IND,
                                :END :END_IND;
```

You can declare these variables as follows:

```
DCL CLS_CD CHAR(7);
DCL DAY BIN FIXED(15);
DCL BGN CHAR(8);
DCL END CHAR(8);
DCL (DAY_IND, BGN_IND, END_IND) BIN FIXED(15);
```

Related concepts:

“Indicator variables, arrays, and structures” on page 176

Related tasks:

“Inserting null values into columns by using indicator variables or arrays” on page 190

Equivalent SQL and PL/I data types

When you declare host variables in your PL/I programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base `SQLTYPE` and `SQLLEN` values that the precompiler uses for host variables in SQL statements.

Table 71. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in PL/I programs

PL/I host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
BIN FIXED(<i>n</i>) 1<= <i>n</i> <=15	500	2	SMALLINT
BIN FIXED(<i>n</i>) 16<= <i>n</i> <=31	496	4	INTEGER
FIXED BIN(63)	492	8	BIGINT
DEC FIXED(<i>p,s</i>) 0<= <i>p</i> <=31 and 0<= <i>s</i> <= <i>p</i> ²	484	<i>p</i> in byte 1, <i>s</i> in byte 2	DECIMAL(<i>p,s</i>)
DEC FLOAT (<i>p</i>) where 1 <= <i>p</i> <= 7	996/997	4	DECFLOAT(16) ⁶
DEC FLOAT (<i>p</i>) where 8 <= <i>p</i> <= 16	996/997	8	DECFLOAT(16)
DEC FLOAT (<i>p</i>) where 17 <= <i>p</i>	996/997	16	DECFLOAT(34)
BIN FLOAT(<i>p</i>) 1<= <i>p</i> <=21	480	4	REAL or FLOAT(<i>n</i>) 1<= <i>n</i> <=21
BIN FLOAT(<i>p</i>) 22<= <i>p</i> <=53	480	8	DOUBLE PRECISION or FLOAT(<i>n</i>) 22<= <i>n</i> <=53
DEC FLOAT(<i>m</i>) 1<= <i>m</i> <=6	480	4	FLOAT (single precision)
DEC FLOAT(<i>m</i>) 7<= <i>m</i> <=16	480	8	FLOAT (double precision)
CHAR(<i>n</i>)	452	<i>n</i>	CHAR(<i>n</i>)
CHAR(<i>n</i>) VARYING 1<= <i>n</i> <=255	448	<i>n</i>	VARCHAR(<i>n</i>)
CHAR(<i>n</i>) VARYING <i>n</i> >255	456	<i>n</i>	VARCHAR(<i>n</i>)
GRAPHIC(<i>n</i>)	468	<i>n</i>	GRAPHIC(<i>n</i>)
GRAPHIC VARYING(<i>n</i>)	464	<i>n</i>	VARGRAPHIC(<i>n</i>)

Table 71. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in PL/I programs (continued)

PL/I host variable data type	SQLTYPE of host variable ¹	SQLLEN of host variable	SQL data type
SQL TYPE IS BINARY(<i>n</i>), 1<= <i>n</i> <=255	912	<i>n</i>	BINARY(<i>n</i>)
SQL TYPE IS VARBINARY(<i>n</i>), 1<= <i>n</i> <=32 704	908	<i>n</i>	VARBINARY(<i>n</i>)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator ³
SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	976	4	Table locator ³
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB locator ³
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB locator ³
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB locator ³
SQL TYPE IS BLOB(<i>n</i>) 1≤ <i>n</i> ≤2147483647	404	<i>n</i>	BLOB(<i>n</i>)
SQL TYPE IS CLOB(<i>n</i>) 1≤ <i>n</i> ≤2147483647	408	<i>n</i>	CLOB(<i>n</i>)
SQL TYPE IS DBCLOB(<i>n</i>) 1≤ <i>n</i> ≤1073741823 ⁴	412	<i>n</i>	DBCLOB(<i>n</i>) ⁴
SQL TYPE IS XML AS BLOB(<i>n</i>)	404	0	XML
SQL TYPE IS XML AS CLOB(<i>n</i>)	408	0	XML
SQL TYPE IS XML AS DBCLOB(<i>n</i>)	412	0	XML
SQL TYPE IS BLOB_FILE	916/917	267	BLOB file reference ³
SQL TYPE IS CLOB_FILE	920/921	267	CLOB file reference ³
SQL TYPE IS DBCLOB_FILE	924/925	267	DBCLOB file reference ³
SQL TYPE IS XML AS BLOB_FILE	916/917	267	XML BLOB file reference ³
SQL TYPE IS XML AS CLOB_FILE	920/921	267	XML CLOB file reference ³
SQL TYPE IS XML AS DBCLOB_FILE	924/925	267	XML DBCLOB file reference ³
SQL TYPE IS ROWID	904	40	ROWID
WIDECHAR(<i>n</i>)	468	<i>n</i>	GRAPHIC(<i>n</i>) ⁵
WIDECHAR VARYING(<i>n</i>)	464	<i>n</i>	VARGRAPHIC(<i>n</i>) ⁵

Notes:

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
2. If *p*=0, DB2 interprets it as DECIMAL(31). For example, DB2 interprets a PL/I data type of DEC FIXED(0,0) to be DECIMAL(31,0), which equates to the SQL data type of DECIMAL(31,0).
3. Do not use this data type as a column type.
4. *n* is the number of double-byte characters.
- | 5. CCSID 1200 is always assigned to WIDECHAR type host var.
- | 6. The data type conversions can be used only if the DB2 coprocessor is used, and the PL/I compiler options
| FLOAT(DFP) and ARCH(7) are specified.

The following table shows equivalent PL/I host variables for each SQL data type. Use this table to determine the PL/I data type for host variables that you define to receive output from the database. For example, if you retrieve `TIMESTAMP` data, you can define a variable of type `CHAR(n)`.

This table shows direct conversions between SQL data types and PL/I data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, DB2 converts those compatible data types.

Table 72. PL/I host variable equivalents that you can use when retrieving data of a particular SQL data type

SQL data type	PL/I host variable equivalent	Notes
SMALLINT	BIN FIXED(<i>n</i>)	1 <= <i>n</i> <= 15
INTEGER	BIN FIXED(<i>n</i>)	16 <= <i>n</i> <= 31
BIGINT	FIXED BIN(63)	
DECIMAL(<i>p,s</i>) or NUMERIC(<i>p,s</i>)	If <i>p</i> < 16: DEC FIXED(<i>p</i>) or DEC FIXED(<i>p,s</i>)	<i>p</i> is precision; <i>s</i> is scale. 1 <= <i>p</i> <= 31 and 0 <= <i>s</i> <= <i>p</i> If <i>p</i> > 15, the PL/I compiler must support 31-digit decimal variables.
DECFLOAT(16)	DEC FLOAT (<i>p</i>)	1 <= <i>p</i> <= 7
DECFLOAT(16)	DEC FLOAT (<i>p</i>)	8 <= <i>p</i> <= 16
DECFLOAT(34)	DEC FLOAT (<i>p</i>)	17 <= <i>p</i>
REAL or FLOAT(<i>n</i>)	BIN FLOAT(<i>p</i>) or DEC FLOAT(<i>m</i>)	1 <= <i>n</i> <= 21, 1 <= <i>p</i> <= 21, and 1 <= <i>m</i> <= 6
DOUBLE PRECISION, DOUBLE, or FLOAT(<i>n</i>)	BIN FLOAT(<i>p</i>) or DEC FLOAT(<i>m</i>)	22 <= <i>n</i> <= 53, 22 <= <i>p</i> <= 53, and 7 <= <i>m</i> <= 16
CHAR(<i>n</i>)	CHAR(<i>n</i>)	1 <= <i>n</i> <= 255
VARCHAR(<i>n</i>)	CHAR(<i>n</i>) VAR	
GRAPHIC(<i>n</i>)	GRAPHIC(<i>n</i>) or WIDECHAR(<i>n</i>) ²	<i>n</i> refers to the number of double-byte characters, not to the number of bytes.
VARGRAPHIC(<i>n</i>)	GRAPHIC(<i>n</i>) VARYING or WIDECHAR(<i>n</i>) VARYING	<i>n</i> refers to the number of double-byte characters, not to the number of bytes.
BINARY(<i>n</i>)	SQL TYPE IS BINARY(<i>n</i>)	1 <= <i>n</i> <= 255
VARBINARY(<i>n</i>)	SQL TYPE IS VARBINARY(<i>n</i>)	1 <= <i>n</i> <= 32 704
DATE	CHAR(<i>n</i>)	If you are using a date exit routine, that routine determines <i>n</i> ; otherwise, <i>n</i> must be at least 10.
TIME	CHAR(<i>n</i>)	If you are using a time exit routine, that routine determines <i>n</i> . Otherwise, <i>n</i> must be at least 6; to include seconds, <i>n</i> must be at least 8.
TIMESTAMP	CHAR(<i>n</i>)	<i>n</i> must be at least 19. To include microseconds, <i>n</i> must be 26; if <i>n</i> is less than 26, the microseconds part is truncated.
TIMESTAMP(0)	CHAR(<i>n</i>)	<i>n</i> must be at least 19.
TIMESTAMP(<i>p</i>) <i>p</i> > 0	CHAR(<i>n</i>)	<i>n</i> must be at least 19. To include fractional seconds, <i>n</i> must be 20+ <i>x</i> where <i>x</i> is the number of fractional seconds to include; if <i>x</i> is less than <i>p</i> , truncation occurs on the fractional seconds part.

Table 72. PL/I host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	PL/I host variable equivalent	Notes
TIMESTAMP(0) WITH TIME ZONE	CHAR(<i>n</i>) VAR	<i>n</i> must be at least 25.
TIMESTAMP(<i>p</i>) WITH TIME ZONE	CHAR(<i>n</i>) VAR	<i>n</i> must be at least 26+ <i>p</i> .
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type. ²
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type. ²
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type. ²
BLOB(<i>n</i>)	SQL TYPE IS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647 ²
CLOB(<i>n</i>)	SQL TYPE IS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647 ²
DBCLOB(<i>n</i>)	SQL TYPE IS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823 ¹
XML	SQL TYPE IS XML AS BLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS CLOB(<i>n</i>)	1≤ <i>n</i> ≤2147483647
XML	SQL TYPE IS XML AS DBCLOB(<i>n</i>)	<i>n</i> is the number of double-byte characters. 1≤ <i>n</i> ≤1073741823 ²
BLOB file reference	SQL TYPE IS BLOB_FILE	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type. ²
CLOB file reference	SQL TYPE IS CLOB_FILE	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type. ²
DBCLOB file reference	SQL TYPE IS DBCLOB_FILE	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type. ²
XML BLOB file reference	SQL TYPE IS XML AS BLOB_FILE	Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.
XML CLOB file reference	SQL TYPE IS XML AS CLOB_FILE	Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.
XML DBCLOB file reference	SQL TYPE IS XML AS DBCLOB_FILE	Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.
ROWID	SQL TYPE IS ROWID	

Table 72. PL/I host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

SQL data type	PL/I host variable equivalent	Notes
Notes:		
1.	CCSID 1200 is always assigned to WIDECHAR type host var.	
2.	The data type conversions can be used only if the DB2 coprocessor is used, and the PL/I compiler options FLOAT(DFP) and ARCH(7) are specified.	

Related concepts:

“Compatibility of SQL and language data types” on page 180

“LOB host variable, LOB locator, and LOB file reference variable declarations” on page 757

“Host variable data types for XML data in embedded SQL applications” on page 241

SQL statements in PL/I programs

You can code SQL statements in a PL/I program wherever you can use executable statements.

The first statement of the PL/I program must be the PROCEDURE statement with OPTIONS(MAIN), unless the program is a stored procedure. A stored procedure application can run as a subroutine.

Each SQL statement in a PL/I program must begin with EXEC SQL and end with a semicolon (;). The EXEC and SQL keywords must appear must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in a PL/I program as follows:

```
EXEC SQL UPDATE DSN8A10.DEPT
        SET MGRNO = :MGR_NUM
        WHERE DEPTNO = :INT_DEPT ;
```

Comments: You can include PL/I comments in embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can also include SQL comments in any SQL statement.

To include DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string.

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for other PL/I statements, except that you must specify EXEC SQL on one line.

Declaring tables and views: Your PL/I program should include a DECLARE TABLE statement to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements.

Including code: You can use SQL statements or PL/I host variable declarations from a member of a partitioned data set by using the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use the PL/I %INCLUDE statement to include SQL statements or host variable DCL statements. You must use the PL/I preprocessor to resolve any %INCLUDE statements before you use the DB2 precompiler. Do not use PL/I preprocessor directives within SQL statements.

Margins: Code SQL statements in columns 2 through 72, unless you have specified other margins to the DB2 precompiler. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid PL/I name for a host variable. Do not use external entry names or access plan names that begin with 'DSN', and do not use host variable names that begin with 'SQL'. These names are reserved for DB2.

Sequence numbers: The source statements that the DB2 precompiler generates do not include sequence numbers. IEL0378I messages from the PL/I compiler identify lines of code without sequence numbers. You can ignore these messages.

Statement labels: You can specify a statement label for executable SQL statements. However, the INCLUDE *text-file-name* and END DECLARE SECTION statements cannot have statement labels.

Whenever statement: The target for the GOTO clause in an SQL statement WHENEVER must be a label in the PL/I source code and must be within the scope of any SQL statements that WHENEVER affects.

Using double-byte character set (DBCS) characters: The following considerations apply to using DBCS in PL/I programs with SQL statements:

- If you use DBCS in the PL/I source, DB2 rules for the following language elements apply:
 - Graphic strings
 - Graphic string constants
 - Host identifiers
 - Mixed data in character strings
 - MIXED DATA option
- The PL/I preprocessor transforms the format of DBCS constants. If you do not want that transformation, run the DB2 precompiler **before** the preprocessor.
- If you use graphic string constants or mixed data in dynamically prepared SQL statements, and if your application requires the PL/I Version 2 (or later) compiler, the dynamically prepared statements must use the PL/I mixed constant format.
 - If you prepare the statement from a host variable, change the string assignment to a PL/I mixed string.
 - If you prepare the statement from a PL/I string, change that to a host variable, and then change the string assignment to a PL/I mixed string.

Example:

```
SQLSTMT = 'SELECT <dbdb> FROM table-name'M;  
EXEC SQL PREPARE STMT FROM :SQLSTMT;
```

- If you want a DBCS identifier to resemble a PL/I graphic string, you must use a delimited identifier.
- If you include DBCS characters in comments, you must delimit the characters with a shift-out and shift-in control character. The first shift-in character signals the end of the DBCS string.

- You can declare host variable names that use DBCS characters in PL/I application programs. The rules for using DBCS variable names in PL/I follow existing rules for DBCS SQL ordinary identifiers, except for length. The maximum length for a host variable is 128 Unicode bytes in DB2. For information about the rules for DBCS SQL ordinary identifiers, see the information about SQL identifiers.

Restrictions:

- DBCS variable names must contain DBCS characters only. Mixing single-byte character set (SBCS) characters with DBCS characters in a DBCS variable name produces unpredictable results.
- A DBCS variable name cannot continue to the next line.
- The PL/I preprocessor changes non-Kanji DBCS characters into extended binary coded decimal interchange code (EBCDIC) SBCS characters. To avoid this change, use Kanji DBCS characters for DBCS variable names, or run the PL/I compiler without the PL/I preprocessor.

Special PL/I considerations: The following considerations apply to programs written in PL/I:

- When compiling a PL/I program that includes SQL statements, you must use the PL/I compiler option CHARSET (60 EBCDIC).
- In unusual cases, the generated comments in PL/I can contain a semicolon. The semicolon generates compiler message IEL0239I, which you can ignore.
- The generated code in a PL/I declaration can contain the ADDR function of a field defined as character varying. This produces either message IBM105I I or IBM1180I W, both of which you can ignore.
- The precompiler generated code in PL/I source can contain the NULL() function. This produces message IEL0533I, which you can ignore unless you also use NULL as a PL/I variable. If you use NULL as a PL/I variable in a DB2 application, you must also declare NULL as a built-in function (DCL NULL BUILTIN;) to avoid PL/I compiler errors.
- The PL/I macro processor can generate SQL statements or host variable DCL statements if you run the macro processor before running the DB2 precompiler. If you use the PL/I macro processor, do not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the COPTION parameter of the DSNH command or the option PARM.PLI=*options* of the EXEC statement in the DSNHPLI procedure.
- Using the PL/I multitasking facility, in which multiple tasks execute SQL statements, causes unpredictable results.
- PL/I WIDECHAR host data type is supported through the DB2 coprocessor only.
- When you use PL/I WX widechar constant, DB2 supports only bigendian format. Thus, when you assign a constant to the widechar type host variable in PL/I, ensure that bigendian format is used. For example:

```
HVWC1 = '003100320033006100620063'WX;
```

Equivalent to:

```
HVWC1 = '123abc';
```

HVWC1 is defined as a WIDECHAR type host variable.

- PL/I SQL Preprocessor option, CCSID0 and NOCCSID0, usage consideration when used with the DB2 coprocessor.

- When you use CCSID0 (default), it promotes compatibility with older PL/I programs, which used the DB2 precompiler. During program preparation, no CCSID value is associated with the host variable except for the WIDECHAR type host variable. For WIDECHAR type host variable, CCSID 1200 is always assigned by the PL/I SQL Preprocessor.

During BIND and runtime, if no CCSID is associated with the host variable, the BIND option, ENCODING, which is meant for the application data, is used. If the ENCODING BIND option is not specified, then the default value for the ENCODING BIND option is used.

- When you use NOCCSID0, a CCSID is associated with the host variable during program preparation. The CCSID is derived from the following items during program preparation:

- DECLARE :hv VARIABLE CCSID xxxx specified.

- Source CCSID, if no DECLARE VARIABLE ... CCSID xxxx is specified for the host variable. During BIND time, note the CCSID assigned to the host variable during program preparation is not known to the BIND process. For more information about BIND time CCSID resolution, see Determining the encoding scheme and CCSID of a string (DB2 SQL).

For host variable used in static SQL, ensuring accurate and matching CCSID is assigned/derived through DECLARE VARIABLE ... CCSID xxxx, source CCSID or ENCODING BIND option or the installation default

For parameter marker used in dynamic SQL, ensuring accurate CCSID for the corresponding host variable is assigned/derived through DECLARE VARIABLE ... CCSID xxxx, ENCODING BIND option or the installation default. The source CCSID has no influence on parameter marker.

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 229.

You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234.

DSNTIAR syntax:

```
CALL DSNTIAR ( sqlca, message, lrecl );
```

The DSNTIAR parameters have the following meanings:

sqlca

An SQL communication area.

message

An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:


```

DCL DATA_LEN FIXED BIN(31) INIT(132);
DCL DATA_DIM FIXED BIN(31) INIT(10);
DCL 1 ERROR_MESSAGE AUTOMATIC,
      3 ERROR_LEN    FIXED BIN(15) UNAL INIT((DATA_LEN*DATA_DIM)),
      3 ERROR_TEXT(DATA_DIM) CHAR(DATA_LEN);
:
CALL DSNTIAR ( SQLCA, ERROR_MESSAGE, DATA_LEN );

```

where `ERROR_MESSAGE` is the name of the message output area, `DATA_DIM` is the number of lines in the message output area, and `DATA_LEN` is the length of each line.

lrecl

A fullword containing the logical record length of output messages, between 72 and 240.

Because `DSNTIAR` is an assembler language program, you must include the following directives in your PL/I application:

```
DCL DSNTIAR ENTRY OPTIONS (ASM,INTER,RETCODE);
```

An example of calling `DSNTIAR` from an application appears in the DB2 sample assembler program `DSN8BP3`, contained in the library `DSN8A10.SDSNSAMP`. See “DB2 sample applications” on page 1092 for instructions on how to access and print the source code for the sample program.

CICS: If your CICS application requires CICS storage handling, you must use the subroutine `DSNTIAC` instead of `DSNTIAR`. `DSNTIAC` has the following syntax:

```
CALL DSNTIAC (eib, commarea, sqlca, msg, lrecl);
```

`DSNTIAC` has extra parameters, which you must use for calls to routines that use CICS commands.

eib EXEC interface block

commarea
communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for `DSNTIAR`. Both `DSNTIAC` and `DSNTIAR` format the `SQLCA` in the same way.

You must define `DSNTIA1` in the CSD. If you load `DSNTIAR` or `DSNTIAC`, you must also define them in the CSD. For an example of CSD entry generation statements for use with `DSNTIAC`, see job `DSNTEJ5A`.

The assembler source code for `DSNTIAC` and job `DSNTEJ5A`, which assembles and link-edits `DSNTIAC`, are in the data set *prefix*.`SDSNSAMP`.

Related concepts:

“DCLGEN (declarations generator)” on page 161

“Host variable arrays in an SQL statement” on page 191

 SQL identifiers (DB2 SQL)

Related tasks:

“Including dynamic SQL in your program” on page 193

“Embedding SQL statements in your application” on page 183

“Handling SQL error codes” on page 239

 Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Delimiters in SQL statements in PL/I programs

You must delimit SQL statements in your PL/I program so that DB2 knows when a particular SQL statement ends.

Delimit an SQL statement in your PL/I program with the beginning keyword EXEC SQL and a Semicolon (;).

Programming examples in PL/I

You can write DB2 programs in PL/I. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in DSN910.SDSNSAMP as a model for your JCL.

Related reference:

“Programming examples” on page 251

Example PL/I program that calls a stored procedure

You can call the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention from a PL/I program on a z/OS system.

The following figure contains the example PL/I program that calls the GETPRML stored procedure.

```

*PROCESS SYSTEM(MVS);
CALPRML:
  PROC OPTIONS(MAIN);

  /*****
  /* Declare the parameters used to call the GETPRML
  /* stored procedure.
  *****/
  DECLARE PROCNM CHAR(18), /* INPUT parm -- PROCEDURE name */
          SCHEMA CHAR(8), /* INPUT parm -- User's schema */
          OUT_CODE FIXED BIN(31),
          /* OUTPUT -- SQLCODE from the
          /* SELECT operation.
          PARMNST CHAR(254) /* OUTPUT -- RUNOPTS for
          VARYING, /* the matching row in the
          /* catalog table SYSROUTINES
          PARMIND FIXED BIN(15);
          /* PARMNST indicator variable
  *****/
  /* Include the SQLCA
  *****/
  EXEC SQL INCLUDE SQLCA;
  /*****
  /* Call the GETPRML stored procedure to retrieve the
  /* RUNOPTS values for the stored procedure. In this
  /* example, we request the RUNOPTS values for the
  /* stored procedure named DSN8EP2.
  *****/
  PROCNM = 'DSN8EP2';
          /* Input parameter -- PROCEDURE to be found
  SCHEMA = ' ';
          /* Input parameter -- SCHEMA in SYSROUTINES
  PARMIND = -1; /* The PARMNST parameter is an output parm.
          /* Mark PARMNST parameter as null, so the DB2
          /* requester does not have to send the entire
          /* PARMNST variable to the server. This
          /* helps reduce network I/O time, because
          /* PARMNST is fairly large.
  EXEC SQL
  CALL GETPRML(:PROCNM,
              :SCHEMA,
              :OUT_CODE,
              :PARMNST INDICATOR :PARMIND);
  IF SQLCODE<=0 THEN /* If SQL CALL failed,
  DO;
  PUT SKIP EDIT('SQL CALL failed due to SQLCODE = ',
              SQLCODE) (A(34),A(14));
  PUT SKIP EDIT('SQLERRM = ',
              SQLERRM) (A(10),A(70));
  END;
  ELSE /* If the CALL worked,
  IF OUT_CODE<=0 THEN /* Did GETPRML hit an error?
  PUT SKIP EDIT('GETPRML failed due to RC = ',
              OUT_CODE) (A(33),A(14));
  ELSE /* Everything worked.
  PUT SKIP EDIT('RUNOPTS = ', PARMNST) (A(11),A(200));
  RETURN;
  END CALPRML;

```

Figure 22. Calling a stored procedure from a PL/I program

Example PL/I stored procedure with a GENERAL linkage convention

You can call a stored procedure that uses the GENERAL linkage convention from a PL/I program.

This example stored procedure searches the DB2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.

The linkage convention for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
  OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
  LANGUAGE PLI
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME "GETPRML"
  COLLID GETPRML
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL
  STAY RESIDENT NO
  RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
  WLM ENVIRONMENT SAMPPROG
  PROGRAM TYPE MAIN
  SECURITY DB2
  RESULT SETS 0
  COMMIT ON RETURN NO;
```

The following example is a PL/I stored procedure with linkage convention GENERAL.

```
*PROCESS SYSTEM(MVS);

GETPRML:
  PROC(PROCNM, SCHEMA, OUT_CODE, PARMLST)
    OPTIONS(MAIN NOEXECOPS REENTRANT);

  DECLARE PROCNM CHAR(18), /* INPUT parm -- PROCEDURE name */
          SCHEMA CHAR(8), /* INPUT parm -- User's SCHEMA */

          OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from */
                                /* the SELECT operation. */
          PARMLST CHAR(254) /* OUTPUT -- RUNOPTS for */
          VARYING; /* the matching row in */
                                /* SYSIBM.SYSROUTINES */

  EXEC SQL INCLUDE SQLCA;

  /*****
  /* Execute SELECT from SYSIBM.SYSROUTINES in the catalog. */
  *****/
  EXEC SQL
    SELECT RUNOPTS INTO :PARMLST
    FROM SYSIBM.SYSROUTINES
    WHERE NAME=:PROCNM AND
          SCHEMA=:SCHEMA;
```

```

    OUT_CODE = SQLCODE;          /* return SQLCODE to caller */
    RETURN;
END GETPRML;

```

Example PL/I stored procedure with a GENERAL WITH NULLS linkage convention

You can call a stored procedure that uses the GENERAL WITH NULLS linkage convention from a PL/I program.

This example stored procedure searches the DB2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```

CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
    OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE PLI
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 0
COMMIT ON RETURN NO;

```

The following example is a PL/I stored procedure with linkage convention GENERAL WITH NULLS.

```
*PROCESS SYSTEM(MVS);
```

```

GETPRML:
  PROC(PROCNM, SCHEMA, OUT_CODE, PARMLST, INDICATORS)
    OPTIONS(MAIN NOEXECOPS REENTRANT);

  DECLARE PROCNM CHAR(18),    /* INPUT parm -- PROCEDURE name */
          SCHEMA CHAR(8),    /* INPUT parm -- User's schema */

          OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from */
                                   /* the SELECT operation. */
          PARMLST CHAR(254)    /* OUTPUT -- PARMLIST for */
          VARYING;           /* the matching row in */
                                   /* SYSIBM.SYSROUTINES */
  DECLARE 1 INDICATORS,      /* Declare null indicators for */
                                   /* input and output parameters. */
          3 PROCNM_IND  FIXED BIN(15),
          3 SCHEMA_IND  FIXED BIN(15),
          3 OUT_CODE_IND FIXED BIN(15),
          3 PARMLST_IND  FIXED BIN(15);

```

```

EXEC SQL INCLUDE SQLCA;

IF PROCNM_IND<0 |
  SCHEMA_IND<0 THEN
  DO;                                /* If any input parm is NULL, */
    OUT_CODE = 9999;                 /* Set output return code. */
    OUT_CODE_IND = 0;

    /* Output return code is not NULL.*/
    PARMLST_IND = -1;               /* Assign NULL value to PARMLST. */
  END;
ELSE                                  /* If input parms are not NULL, */
  DO;                                  /* */
  /*****
  /* Issue the SQL SELECT against the SYSIBM.SYSROUTINES */
  /* DB2 catalog table. */
  /*****
  EXEC SQL
    SELECT RUNOPTS INTO :PARMLST
      FROM SYSIBM.SYSROUTINES
      WHERE NAME=:PROCNM AND
            SCHEMA=:SCHEMA;
    PARMLST_IND = 0;                 /* Mark PARMLST as not NULL. */

    OUT_CODE = SQLCODE;             /* return SQLCODE to caller */
    OUT_CODE_IND = 0;
    OUT_CODE_IND = 0;               /* Output return code is not NULL.*/
  END;
RETURN;

END GETPRML;

```

Chapter 9. Coding SQL statements in REXX application programs

When you code SQL statements in REXX application programs, you should follow certain guidelines.

Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX

When DB2 prepares a REXX program that contains SQL statements, DB2 automatically includes an SQLCA in the program.

The REXX SQLCA differs from the SQLCA for other languages. The REXX SQLCA consists of a set of separate variables, rather than a structure.

The SQLCA has the following forms:

- A set of simple variables
- A set of compound variables that begin with the stem SQLCA

The simple variables is the default form of the SQLCA. Using CALL SQLEXEC results in the compound stem variables. Otherwise, the attachment command used determines the form of the SQLCA. If you use the ADDRESS DSNREXX 'CONNECT' *ssid* syntax to connect to DB2, the SQLCA variables are a set of simple variables. If you use the CALL SQLDBS 'ATTACH TO' syntax to connect to DB2, the SQLCA variables are compound variables that begin with the stem SQLCA.

Switching forms of the SQLCA within an application is not recommended.

Related tasks:

“Checking the execution of SQL statements” on page 227

“Checking the execution of SQL statements by using the SQLCA” on page 228

“Checking the execution of SQL statements by using SQLCODE and SQLSTATE” on page 232

“Defining the items that your program can use to check whether an SQL statement executed successfully” on page 173

Defining SQL descriptor areas in REXX

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or DB2.

To define SQL descriptor areas:

Code the SQLDA declarations directly in your program.

Each SQLDA consists of a set of REXX variables with a common stem. The stem must be a REXX variable name that contains no periods and is the same as the value of *descriptor-name* that you specify when you use the SQLDA in an SQL statement.

Restrictions:

- You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.
- You cannot use the SQL INCLUDE statement for the SQLDA, because it is not supported in COBOL.

Related tasks:

“Defining SQL descriptor areas” on page 173

Equivalent SQL and REXX data types

All REXX data is string data. Therefore, when a REXX program assigns input data to a column, DB2 converts the data from a string type to the column type. When a REXX program assigns column data to an output variable, DB2 converts the data from the column type to a string type.

When you assign input data to a DB2 table column, you can either let DB2 determine the type that your input data represents, or you can use an SQLDA to tell DB2 the intended type of the input data.

When a REXX program assigns data to a column, it can either let DB2 determine the data type or use an SQLDA to specify the intended data type. If the program lets DB2 assign a data type for the input data, DB2 bases its choice on the input string format.

The following table shows the SQL data types that DB2 assigns to input data and the corresponding formats for that data. The two SQLTYPE values that are listed for each data type are the value for a column that does not accept null values and the value for a column that accepts null values.

Table 73. SQL input data types and REXX data formats

SQL data type assigned by DB2	SQLTYPE for data type	REXX input data format
INTEGER	496/497	A string of numerics that does not contain a decimal point or exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -2147483648 and 2147483647, inclusive.
BIGINT	492/493	A string of numbers that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -9223372036854775808 and -2147483648, inclusive, or between 2147483648 and 9223372036854775807.
DECIMAL(<i>p,s</i>)	484/485	One of the following formats: <ul style="list-style-type: none"> • A string of numerics that contains a decimal point but no exponent identifier. <i>p</i> represents the precision and <i>s</i> represents the scale of the decimal number that the string represents. The first character can be a plus (+) or minus (-) sign. • A string of numerics that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented is less than -9223372036854775808 or greater than 9223372036854775807.
FLOAT	480/481	A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus (+) or minus (-) sign and a series of numerics). The string can begin with a plus (+) or minus (-) sign.

Table 73. SQL input data types and REXX data formats (continued)

SQL data type assigned by DB2	SQLTYPE for data type	REXX input data format
VARCHAR(<i>n</i>)	448/449	<p>One of the following formats:</p> <ul style="list-style-type: none"> • A string of length <i>n</i>, enclosed in single or double quotation marks. • The character X or x, followed by a string enclosed in single or double quotation marks. The string within the quotation marks has a length of 2*<i>n</i> bytes and is the hexadecimal representation of a string of <i>n</i> characters. • A string of length <i>n</i> that does not have a numeric or graphic format, and does not satisfy either of the previous conditions.
VARGRAPHIC(<i>n</i>)	464/465	<p>One of the following formats:</p> <ul style="list-style-type: none"> • The character G, g, N, or n, followed by a string enclosed in single or double quotation marks. The string within the quotation marks begins with a shift-out character (X'0E') and ends with a shift-in character (X'0F'). Between the shift-out character and shift-in character are <i>n</i> double-byte characters. • The characters GX, Gx, gX, or gx, followed by a string enclosed in single or double quotation marks. The string within the quotation marks has a length of 4*<i>n</i> bytes and is the hexadecimal representation of a string of <i>n</i> double-byte characters.

For example, when DB2 executes the following statements to update the MIDINIT column of the EMP table, DB2 must determine a data type for HVMIDINIT:

```
SQLSTMT="UPDATE EMP" ,
      "SET MIDINIT = ?" ,
      "WHERE EMPNO = '000200'"
"EXECSQL PREPARE S100 FROM :SQLSTMT"
HVMIDINIT='H'
"EXECSQL EXECUTE S100 USING" ,
      ":HVMIDINIT"
```

Because the data that is assigned to HVMIDINIT has a format that fits a character data type, DB2 REXX Language Support assigns a VARCHAR type to the input data.

If you do not assign a value to a host variable before you assign the host variable to a column, DB2 returns an error code.

Related concepts:

“Compatibility of SQL and language data types” on page 180

SQL statements in REXX programs

You can code SQL statements in a REXX programs wherever you can use REXX commands.

DB2 REXX Language Support supports all dynamic SQL statements and the following static SQL statements:

- CALL
- CLOSE
- CONNECT
- DECLARE CURSOR
- DESCRIBE *prepared statement or table*
- DESCRIBE CURSOR

- DESCRIBE INPUT
- DESCRIBE PROCEDURE
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- OPEN
- PREPARE
- RELEASE *connection*
- SET CONNECTION
- SET CURRENT PACKAGE PATH
- SET CURRENT PACKAGESET
- SET *host-variable* = CURRENT DATE
- SET *host-variable* = CURRENT DEGREE
- SET *host-variable* = CURRENT MEMBER
- SET *host-variable* = CURRENT PACKAGESET
- SET *host-variable* = CURRENT PATH
- SET *host-variable* = CURRENT SERVER
- SET *host-variable* = CURRENT SQLID
- SET *host-variable* = CURRENT TIME
- SET *host-variable* = CURRENT TIMESTAMP
- SET *host-variable* = CURRENT TIMEZONE

Each SQL statement in a REXX program must begin with EXECSQL, in either upper-, lower-, or mixed-case. One of the following items must follow EXECSQL:

- An SQL statement enclosed in single or double quotation marks.
- A REXX variable that contains an SQL statement. The REXX variable must not be preceded by a colon.

For example, you can use either of the following methods to execute the COMMIT statement in a REXX program:

```
EXECSQL "COMMIT"
rexvar="COMMIT"
EXECSQL rexvar
```

The following dynamic statements must be executed using EXECUTE IMMEDIATE or PREPARE and EXECUTE under DSNREXX:

- DECLARE GLOBAL TEMPORARY TABLE
- SET CURRENT DEBUG MODE
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT QUERY ACCELERATION
- SET CURRENT REFRESH AGE
- SET CURRENT ROUTINE VERSION
- SET SCHEMA

You cannot execute a SELECT, INSERT, UPDATE, MERGE, or DELETE statement that contains host variables. Instead, you must execute PREPARE on the statement, with parameter markers substituted for the host variables, and then use the host variables in an EXECUTE, OPEN, or FETCH statement. See “Host variables” on page 174 for more information.

An SQL statement follows rules that apply to REXX commands. The SQL statement can optionally end with a semicolon and can be enclosed in single or double quotation marks, as in the following example:

```
'EXECSQL COMMIT';
```

Comments: You cannot include REXX comments (`/* ... */`) or SQL comments (`--`) within SQL statements. However, you can include REXX comments anywhere else in the program.

Names: Continuation for SQL statements: SQL statements that span lines follow REXX rules for statement continuation. You can break the statement into several strings, each of which fits on a line, and separate the strings with commas or with concatenation operators followed by commas. For example, either of the following statements is valid:

```
EXECSQL ,
  "UPDATE DSN8A10.DEPT" ,
  "SET MGRNO = '000010'" ,
  "WHERE DEPTNO = 'D11'"
"EXECSQL " || ,
" UPDATE DSN8A10.DEPT " || | ,
" SET MGRNO = '000010'" || | ,
" WHERE DEPTNO = 'D11'"
```

Including code: The EXECSQL INCLUDE statement is not valid for REXX. You therefore cannot include externally defined SQL statements in a program.

Margins: Like REXX commands, SQL statements can begin and end anywhere on a line.

You can use any valid REXX name that does not end with a period as a host variable. However, host variable names should not begin with 'SQL', 'RDI', 'DSN', 'RXSQL', or 'QRW'. Variable names can be at most 64 bytes.

Nulls: A REXX null value and an SQL null value are different. The REXX language has a null string (a string of length 0) and a null clause (a clause that contains only blanks and comments). The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a value. Assigning a REXX null value to a DB2 column does not make the column value null.

Statement labels: You can precede an SQL statement with a label, in the same way that you label REXX commands.

Handling errors and warnings: DB2 does not support the SQL WHENEVER statement in a REXX program. To handle SQL errors and warnings, use the following methods:

- To test for SQL errors or warnings, test the SQLCODE or SQLSTATE value and the SQLWARN. values after each EXECSQL call. This method does not detect errors in the REXX interface to DB2.
- To test for SQL errors or warnings or errors or warnings from the REXX interface to DB2, test the REXX RC variable after each EXECSQL call. The following table lists the values of the RC variable.

You can also use the REXX SIGNAL ON ERROR and SIGNAL ON FAILURE keyword instructions to detect negative values of the RC variable and transfer control to an error routine.

Table 74. REXX return codes after SQL statements


Return code	Meaning
0	No SQL warning or error occurred.
+1	An SQL warning occurred.
-1	An SQL error occurred.
-3	The first token after ADDRESS DSNREXX is in error. For a description of the tokens allowed, see "Accessing the DB2 REXX language support application programming interfaces."

Related tasks:

"Including dynamic SQL in your program" on page 193

"Embedding SQL statements in your application" on page 183

"Handling SQL error codes" on page 239

 Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Delimiters in SQL statements in REXX programs

You must delimit SQL statements in your REXX program so that DB2 knows when a particular SQL statement ends.

Delimit an SQL statement in your REXX program by preceding the statement with EXECSQL. If the statement is in a literal string, enclose it in single or double quotation marks.

Accessing the DB2 REXX language support application programming interfaces

DB2 REXX Language Support includes several application programming interfaces that enable your REXX program to connect to a DB2 subsystem and execute SQL statements.

DB2 REXX Language Support includes the following application programming interfaces:

DSNREXX CONNECT

Identifies the REXX task as a connected user of the specified DB2 subsystem. The DSNREXX plan resources are allocated by establishing an allied thread.

You should not confuse the DSNREXX CONNECT command with the DB2 SQL CONNECT statement.

You must execute the DSNREXX CONNECT command before your REXX program can execute SQL statements. Do not use the DSNREXX CONNECT command from a stored procedure.

A currently connected REXX task must be disconnected before switching to a different DB2 subsystem.

The syntax of the DSNREXX CONNECT command is:

```

ADDRESS DSNREXX 'CONNECT' 'subsystem-ID'
                        REXX-variable

```

Notes:

1. CALL SQLDBS 'ATTACH TO' *ssid* is an alternative to ADDRESS DSNREXX 'CONNECT' *ssid*.
2. The *REXX-variable* or '*subsystem-ID*' string may also be a single member name in a data sharing group or the group attachment name.

DSNREXX EXECSQL

Executes SQL statements in REXX programs.

The syntax of the DSNREXX EXECSQL command is:

```

ADDRESS DSNREXX 'EXECSQL' "SQL-statement"
                        REXX-variable

```

Notes:

1. CALL 'SQLEXEC' "SQL-statement" is an alternative to ADDRESS DSNREXX 'EXECSQL' "SQL-statement".
2. 'EXECSQL' and "SQL-statement" can be enclosed in either single or double quotation marks.

DSNREXX DISCONNECT

Deallocates the DSNREXX plan and removes the REXX task as a connected user of DB2.

You should execute the DSNREXX DISCONNECT command to release resources that are held by DB2. Otherwise resources are not released until the REXX task terminates.

Do not use the DSNREXX DISCONNECT command from a stored procedure.

The syntax of the DSNREXX DISCONNECT command is:

```

ADDRESS DSNREXX 'DISCONNECT'

```

Note: CALL SQLDBS 'DETACH' is an alternative to ADDRESS DSNREXX 'DISCONNECT'.

These application programming interfaces are available through the DSNREXX host command environment. To make DSNREXX available to the application, invoke the RXSUBCOM function. The syntax is:

```

RXSUBCOM ( 'ADD' , 'DSNREXX' , 'DSNREXX' )
          'DELETE'

```

The ADD function adds DSNREXX to the REXX host command environment table. The DELETE function deletes DSNREXX from the REXX host command environment table.

The following figure shows an example of REXX code that makes DSNREXX available to an application.

```
'SUBCOM DSNREXX'                /* HOST CMD ENV AVAILABLE? */
IF RC THEN                      /* IF NOT, MAKE IT AVAILABLE */
  S_RC = RXSUBCOM('ADD', 'DSNREXX', 'DSNREXX')
                                /* ADD HOST CMD ENVIRONMENT */
ADDRESS DSNREXX                 /* SEND ALL COMMANDS OTHER */
                                /* THAN REXX INSTRUCTIONS TO */
                                /* DSNREXX */
                                /* CALL CONNECT, EXECSQL, AND */
                                /* DISCONNECT INTERFACES */
:
:
S_RC = RXSUBCOM('DELETE', 'DSNREXX', 'DSNREXX')
                                /* WHEN DONE WITH */
                                /* DSNREXX, REMOVE IT. */
```

Related concepts:

“REXX stored procedures” on page 650

Ensuring that DB2 correctly interprets character input data in REXX programs

DB2 REXX Language Support might incorrectly interpret character literals as graphic or numeric literals unless you mark them correctly.

To ensure that DB2 correctly interprets character input data in REXX programs:

Precede and follow character literals with a double quotation mark, followed by a single quotation mark, followed by another double quotation mark ("'").

Example: Specify the string the string 100 as "'100'".

Enclosing the string in apostrophes is not adequate, because REXX removes the apostrophes when it assigns a literal to a variable. For example, suppose that you want to pass the value in a host variable called stringvar to DB2. The value that you want to pass is the string '100'. First, you assign the string to the host variable by issuing the following REXX command:

```
stringvar = '100'
```

After the command executes, stringvar contains the characters 100 (without the apostrophes). DB2 REXX Language Support then passes the numeric value 100 to DB2, which is not what you intended.

However, suppose that you write the following command:

```
stringvar = "'100'"
```

In this case, REXX assigns the string '100' to stringvar, including the single quotation marks. DB2 REXX Language Support then passes the string '100' to DB2, which is the result that you want.

Passing the data type of an input data type to DB2 for REXX programs

In certain situations, you should tell DB2 the data type to use for input data in a REXX program. For example, if you are assigning or comparing input data to columns of type SMALLINT, CHAR, or GRAPHIC, you should tell DB2 to use those data types.

DB2 does not assign data types of SMALLINT, CHAR, or GRAPHIC to input data. If you assign or compare this data to columns of type SMALLINT, CHAR, or GRAPHIC, DB2 must do more work than if the data types of the input data and columns match.

To pass the data type of an input data type to DB2 for REXX programs:

Use an SQLDA.

Examples

Example of specifying CHAR as an input data type: Suppose that you want to tell DB2 that the data with which you update the MIDINIT column of the EMP table is of type CHAR, rather than VARCHAR. You need to set up an SQLDA that contains a description of a CHAR column, and then prepare and execute the UPDATE statement using that SQLDA, as shown in the following example.

```
INSQLDA.SQD = 1           /* SQLDA contains one variable */
INSQLDA.1.SQTYPE = 453   /* Type of the variable is CHAR, */
                          /* and the value can be null */
INSQLDA.1.SQLEN = 1     /* Length of the variable is 1 */
INSQLDA.1.SQDATA = 'H'  /* Value in variable is H */
INSQLDA.1.SQLIND = 0    /* Input variable is not null */
SQLSTMT="UPDATE EMP" ,
"SET MIDINIT = ?" ,
"WHERE EMPNO = '000200'"
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING DESCRIPTOR :INSQLDA"
```

Example of specifying the input data type as DECIMAL with precision and scale: Suppose that you want to tell DB2 that the data is of type DECIMAL with precision and nonzero scale. You need to set up an SQLDA that contains a description of a DECIMAL column, as shown in the following example.

```
INSQLDA.SQD = 1           /* SQLDA contains one variable */
INSQLDA.1.SQTYPE = 484   /* Type of variable is DECIMAL */
INSQLDA.1.SQLEN.SQLPRECISION = 18 /* Precision of variable is 18 */
INSQLDA.1.SQLEN.SQLSCALE = 8    /* Scale of variable is 8 */
INSQLDA.1.SQDATA = 9876543210.87654321 /* Value in variable */
```

Setting the isolation level of SQL statements in a REXX program

Isolation levels specify the locking behavior for SQL statements. You can set the isolation level for SQL statements in your REXX program to repeatable read (RR), read stability (RS), cursor stability (CS), or uncommitted read (UR).

To set the isolation level of SQL statements in a REXX program:

Execute the SET CURRENT PACKAGESET statement to select one of the following DB2 REXX Language Support packages with the isolation level that you need.

Table 75. DB2 REXX Language Support packages and associated isolation levels

Package name ^a	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

Note:

1. These packages enable your program to access DB2 and are bound when you install DB2 REXX Language Support.

For example, to change the isolation level to cursor stability, execute the following SQL statement:

```
"EXECSQL SET CURRENT PACKAGESET='DSNREXCS'"
```

Retrieving data from DB2 tables in REXX programs

All output data in REXX programs is string data. Although, you can determine the data type that the data represents from its format and from the data type of the column from which the data was retrieved.

The following table gives the format for each type of output data.

Table 76. SQL output data types and REXX data formats

SQL data type	REXX output data format
SMALLINT INTEGER BIGINT	A string of numerics that does not contain leading zeroes, a decimal point, or an exponent identifier. If the string represents a negative number, it begins with a minus (-) sign. The numeric value is between -9223372036854775808 and 9223372036854775807, inclusive.
DECIMAL(<i>p,s</i>)	A string of numerics with one of the following formats: <ul style="list-style-type: none"> • Contains a decimal point but not an exponent identifier. The string is padded with zeroes to match the scale of the corresponding table column. If the value represents a negative number, it begins with a minus (-) sign. • Does not contain a decimal point or an exponent identifier. The numeric value is less than -9223372036854775808 or greater than 9223372036854775807. If the value is negative, it begins with a minus (-) sign.
FLOAT(<i>n</i>) REAL DOUBLE	A string that represents a number in scientific notation. The string consists of a numeric, a decimal point, a series of numerics, and an exponent identifier. The exponent identifier is an E followed by a minus (-) sign and a series of numerics if the number is between -1 and 1. Otherwise, the exponent identifier is an E followed by a series of numerics. If the string represents a negative number, it begins with a minus (-) sign.
DECFLOAT	REXX emulates the DECFLOAT data type with DOUBLE, so support for DECFLOAT is limited to the REXX support for DOUBLE. The following special values are not supported: <ul style="list-style-type: none"> • INFINITY • SNAN • NAN
CHAR(<i>n</i>) VARCHAR(<i>n</i>)	A character string of length <i>n</i> bytes. The string is not enclosed in single or double quotation marks.
GRAPHIC(<i>n</i>) VARGRAPHIC(<i>n</i>)	A string of length 2* <i>n</i> bytes. Each pair of bytes represents a double-byte character. This string does not contain a leading G, is not enclosed in quotation marks, and does not contain shift-out or shift-in characters.

Because you cannot use the SELECT INTO statement in a REXX procedure, to retrieve data from a DB2 table you must prepare a SELECT statement, open a cursor for the prepared statement, and then fetch rows into host variables or an SQLDA using the cursor. The following example demonstrates how you can retrieve data from a DB2 table using an SQLDA:

```
SQLSTMT= ,
'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,' ,
' WORKDEPT, PHONENO, HIREDATE, JOB,' ,
' EDLEVEL, SEX, BIRTHDATE, SALARY,' ,
' BONUS, COMM' ,
' FROM EMP'
EXECSQL DECLARE C1 CURSOR FOR S1
EXECSQL PREPARE S1 INTO :OUTSQLDA FROM :SQLSTMT
EXECSQL OPEN C1
Do Until(SQLCODE ^= 0)
  EXECSQL FETCH C1 USING DESCRIPTOR :OUTSQLDA
  If SQLCODE = 0 Then Do
    Line = ''
    Do I = 1 To OUTSQLDA.SQLD
      Line = Line OUTSQLDA.I.SQLDATA
    End I
    Say Line
  End
End
```

Cursors and statement names in REXX

In REXX applications that contain SQL statements, you must use a predefined set of names for cursors or prepared statements.

The following names are valid for cursors and prepared statements in REXX applications:

c1 to c100

Cursor names for DECLARE CURSOR, OPEN, CLOSE, and FETCH statements. By default, c1 to c100 are defined with the WITH RETURN clause, and c51 to c100 are defined with the WITH HOLD clause. You can use the ATTRIBUTES clause of the PREPARE statement to override these attributes or add additional attributes. For example, you might want to add attributes to make your cursor scrollable.

c101 to c200

Cursor names for ALLOCATE, DESCRIBE, FETCH, and CLOSE statements that are used to retrieve result sets in a program that calls a stored procedure.

s1 to s100

Prepared statement names for DECLARE STATEMENT, PREPARE, DESCRIBE, and EXECUTE statements.

Use only the predefined names for cursors and statements. When you associate a cursor name with a statement name in a DECLARE CURSOR statement, the cursor name and the statement must have the same number. For example, if you declare cursor c1, you need to declare it for statement s1:

```
EXECSQL 'DECLARE C1 CURSOR FOR S1'
```

Do not use any of the predefined names as host variables names.


Programming examples in REXX

You can write DB2 programs in REXX. These programs can access a local or remote DB2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in DSN910.SDSNSAMP as a model for your JCL.

Related reference:

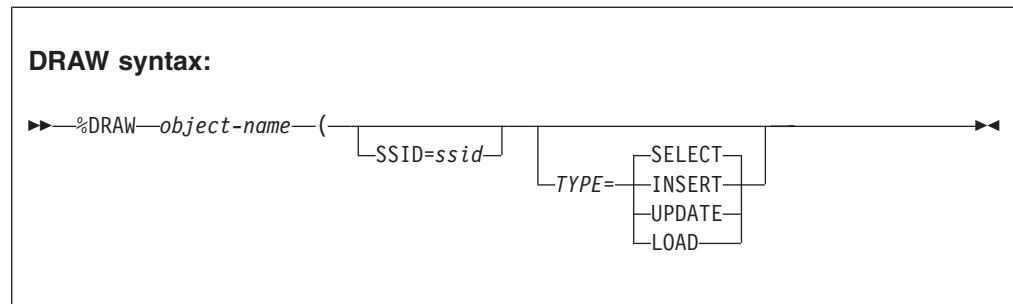
“Programming examples” on page 251

 [DB2 for z/OS Exchange](#)

Sample DB2 REXX application

You can use a REXX application to accept a table name as input and produce a SELECT, INSERT, or UPDATE SQL statement or a LOAD utility statement for the specified table as output.

The following example shows a complete DB2 REXX application named DRAW. DRAW must be invoked from the command line of an ISPF edit session. DRAW takes a table or view name as input and produces a SELECT, INSERT, or UPDATE SQL statement or a LOAD utility control statement that includes the columns of the table as output.



DRAW parameters:

object-name

The name of the table or view for which DRAW builds an SQL statement or utility control statement. The name can be a one-, two-, or three-part name. The table or view to which *object-name* refers must exist before DRAW can run.

object-name is a required parameter.

SSID=ssid

Specifies the name of the local DB2 subsystem.

S can be used as an abbreviation for SSID.

If you invoke DRAW from the command line of the edit session in SPUFI, `SSID=ssid` is an optional parameter. DRAW uses the subsystem ID from the DB2I Defaults panel.

TYPE=operation-type

The type of statement that DRAW builds.

T can be used as an abbreviation for TYPE.

operation-type has one of the following values:

SELECT

Builds a SELECT statement in which the result table contains all columns of *object-name*.

S can be used as an abbreviation for SELECT.

INSERT

Builds a template for an INSERT statement that inserts values into all columns of *object-name*. The template contains comments that indicate where the user can place column values.

I can be used as an abbreviation for INSERT.

UPDATE

Builds a template for an UPDATE statement that updates columns of *object-name*. The template contains comments that indicate where the user can place column values and qualify the update operation for selected rows.

U can be used as an abbreviation for UPDATE.

LOAD

Builds a template for a LOAD utility control statement for *object-name*.

L can be used as an abbreviation for LOAD.

TYPE=*operation-type* is an optional parameter. The default is TYPE=SELECT.

DRAW data sets:

Edit data set

The data set from which you issue the DRAW command when you are in an ISPF edit session. If you issue the DRAW command from a SPUFI session, this data set is the data set that you specify in field 1 of the main SPUFI panel (DSNESP01). The output from the DRAW command goes into this data set.

DRAW return codes:

Return code

Meaning

- | | |
|----|---|
| 0 | Successful completion. |
| 12 | An error occurred when DRAW edited the input file. |
| 20 | One of the following errors occurred: <ul style="list-style-type: none">• No input parameters were specified.• One of the input parameters was not valid.• An SQL error occurred when the output statement was generated. |

Examples of DRAW invocation:

Generate a SELECT statement for table DSN8A10.EMP at the local subsystem. Use the default DB2I subsystem ID.

The DRAW invocation is:

```
DRAW DSN8A10.EMP (TYPE=SELECT
```

The output is:

```
SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,  
      "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,  
      "SALARY" , "BONUS" , "COMM"  
FROM DSN8A10.EMP
```

Generate a template for an INSERT statement that inserts values into table DSN8A10.EMP at location SAN_JOSE. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW SAN_JOSE.DSN8A10.EMP (TYPE=INSERT SSID=DSN
```

The output is:

```
INSERT INTO SAN_JOSE.DSN8A10.EMP ( "EMPNO" , "FIRSTNME" , "MIDINIT" ,
"LASTNAME" , "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" ,
"EDLEVEL" , "SEX" , "BIRTHDATE" , "SALARY" , "BONUS" , "COMM" )
VALUES (
-- ENTER VALUES BELOW      COLUMN NAME      DATA TYPE
, -- EMPNO                   CHAR(6) NOT NULL
, -- FIRSTNME                 VARCHAR(12) NOT NULL
, -- MIDINIT                  CHAR(1) NOT NULL
, -- LASTNAME                 VARCHAR(15) NOT NULL
, -- WORKDEPT                 CHAR(3)
, -- PHONENO                  CHAR(4)
, -- HIREDATE                 DATE
, -- JOB                      CHAR(8)
, -- EDLEVEL                  SMALLINT
, -- SEX                      CHAR(1)
, -- BIRTHDATE               DATE
, -- SALARY                   DECIMAL(9,2)
, -- BONUS                    DECIMAL(9,2)
) -- COMM                     DECIMAL(9,2)
```

Generate a template for an UPDATE statement that updates values of table DSN8A10.EMP. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW DSN8A10.EMP (TYPE=UPDATE SSID=DSN
```

The output is:

```
UPDATE DSN8A10.EMP SET
-- COLUMN NAME      ENTER VALUES BELOW      DATA TYPE
"EMPNO"=           -- CHAR(6) NOT NULL
, "FIRSTNME"=      -- VARCHAR(12) NOT NULL
, "MIDINIT"=       -- CHAR(1) NOT NULL
, "LASTNAME"=      -- VARCHAR(15) NOT NULL
, "WORKDEPT"=      -- CHAR(3)
, "PHONENO"=       -- CHAR(4)
, "HIREDATE"=      -- DATE
, "JOB"=           -- CHAR(8)
, "EDLEVEL"=       -- SMALLINT
, "SEX"=           -- CHAR(1)
, "BIRTHDATE"=     -- DATE
, "SALARY"=        -- DECIMAL(9,2)
, "BONUS"=         -- DECIMAL(9,2)
, "COMM"=          -- DECIMAL(9,2)
WHERE
```

Generate a LOAD control statement to load values into table DSN8A10.EMP. The local subsystem ID is DSN.

The draw invocation is:

```
DRAW DSN8A10.EMP (TYPE=LOAD SSID=DSN
```

The output is:

```

LOAD DATA INDDN SYSREC INTO TABLE DSN8A10.EMP
( "EMPNO"          POSITION(  1) CHAR(6)
, "FIRSTNME"      POSITION(  8) VARCHAR
, "MIDINIT"       POSITION( 21) CHAR(1)
, "LASTNAME"      POSITION( 23) VARCHAR
, "WORKDEPT"     POSITION( 39) CHAR(3)
                  NULLIF( 39)='?'
, "PHONENO"       POSITION( 43) CHAR(4)
                  NULLIF( 43)='?'
, "HIREDATE"      POSITION( 48) DATE EXTERNAL
                  NULLIF( 48)='?'
, "JOB"           POSITION( 59) CHAR(8)
                  NULLIF( 59)='?'
, "EDLEVEL"       POSITION( 68) SMALLINT
                  NULLIF( 68)='?'
, "SEX"           POSITION( 71) CHAR(1)
                  NULLIF( 71)='?'
, "BIRTHDATE"    POSITION( 73) DATE EXTERNAL
                  NULLIF( 73)='?'
, "SALARY"        POSITION( 84) DECIMAL EXTERNAL(9,2)
                  NULLIF( 84)='?'
, "BONUS"         POSITION( 90) DECIMAL EXTERNAL(9,2)
                  NULLIF( 90)='?'
, "COMM"          POSITION( 96) DECIMAL EXTERNAL(9,2)
                  NULLIF( 96)='?'
)

```

DRAW source code:

```

/* REXX *****/
L1 = WHEREAMI()
/*
DRAW creates basic SQL queries by retrieving the description of a
table. You must specify the name of the table or view to be queried.
You can specify the type of query you want to compose. You might need
to specify the name of the DB2 subsystem.
>>--DRAW-----tablename-----|-----<<
                                |-----|
                                |-(|-Ssid=subsystem-name-|-|
                                |   +-Select+
                                |-Type=-|-Insert-|----|
                                |   |Update|
                                |   +---Load---+

```

Ssid=subsystem-name

subsystem-name specified the name of a DB2 subsystem.

Select

Composes a basic query for selecting data from the columns of a table or view. If TYPE is not specified, SELECT is assumed.

Using SELECT with the DRAW command produces a query that would retrieve all rows and all columns from the specified table. You can then modify the query as needed.

A SELECT query of EMP composed by DRAW looks like this:

```

SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,
      "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,
      "SALARY" , "BONUS" , "COMM"
FROM DSN8A10.EMP

```

If you include a location qualifier, the query looks like this:

```

SELECT "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" ,
      "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" ,
      "SALARY" , "BONUS" , "COMM"
FROM STLEC1.DSN8A10.EMP

```

To use this SELECT query, type the other clauses you need. If you are selecting from more than one table, use a DRAW command for each table name you want represented.

Insert

Composes a basic query to insert data into the columns of a table or view.

The following example shows an INSERT query of EMP that

DRAW composed:

```

INSERT INTO DSN8A10.EMP ( "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" ,
    "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" ,
    "BIRTHDATE" , "SALARY" , "BONUS" , "COMM" )
VALUES (
-- ENTER VALUES BELOW      COLUMN NAME      DATA TYPE
, -- EMPNO                   CHAR(6) NOT NULL
, -- FIRSTNME                 VARCHAR(12) NOT NULL
, -- MIDINIT                  CHAR(1) NOT NULL
, -- LASTNAME                 VARCHAR(15) NOT NULL
, -- WORKDEPT                 CHAR(3)
, -- PHONENO                  CHAR(4)
, -- HIREDATE                 DATE
, -- JOB                      CHAR(8)
, -- EDLEVEL                  SMALLINT
, -- SEX                      CHAR(1)
, -- BIRTHDATE               DATE
, -- SALARY                   DECIMAL(9,2)
, -- BONUS                    DECIMAL(9,2)
) -- COMM                     DECIMAL(9,2)

```

To insert values into EMP, type values to the left of the column names.

Update

Composes a basic query to change the data in a table or view. The following example shows an UPDATE query of EMP composed by DRAW:

```

UPDATE DSN8A10.EMP SET
-- COLUMN NAME      ENTER VALUES BELOW      DATA TYPE
"EMPNO"=           -- CHAR(6) NOT NULL
, "FIRSTNME"=      -- VARCHAR(12) NOT NULL
, "MIDINIT"=       -- CHAR(1) NOT NULL
, "LASTNAME"=     -- VARCHAR(15) NOT NULL
, "WORKDEPT"=     -- CHAR(3)
, "PHONENO"=      -- CHAR(4)
, "HIREDATE"=     -- DATE
, "JOB"=          -- CHAR(8)
, "EDLEVEL"=     -- SMALLINT
, "SEX"=          -- CHAR(1)
, "BIRTHDATE"=   -- DATE
, "SALARY"=      -- DECIMAL(9,2)
, "BONUS"=       -- DECIMAL(9,2)
, "COMM"=        -- DECIMAL(9,2)

```

WHERE

To use this UPDATE query, type the changes you want to make to the right of the column names, and delete the lines you do not need. Be sure to complete the WHERE clause.

Load

Composes a load statement to load the data in a table. The following example shows a LOAD statement of EMP composed by DRAW:

```

LOAD DATA INDDN SYSREC INTO TABLE DSN8A10 .EMP
( "EMPNO"          POSITION( 1) CHAR(6)
, "FIRSTNME"      POSITION( 8) VARCHAR
, "MIDINIT"       POSITION( 21) CHAR(1)
, "LASTNAME"      POSITION( 23) VARCHAR
, "WORKDEPT"      POSITION( 39) CHAR(3)
, "PHONENO"       POSITION( 43) CHAR(4)
, "HIREDATE"      POSITION( 48) DATE EXTERNAL
, "JOB"           POSITION( 59) CHAR(8)
, "EDLEVEL"       POSITION( 68) SMALLINT
, "SEX"           POSITION( 71) CHAR(1)

```

```

, "BIRTHDATE"      POSITION( 73) DATE EXTERNAL
                    NULLIF( 73)='?'
, "SALARY"         POSITION( 84) DECIMAL EXTERNAL(9,2)
                    NULLIF( 84)='?'
, "BONUS"          POSITION( 90) DECIMAL EXTERNAL(9,2)
                    NULLIF( 90)='?'
, "COMM"           POSITION( 96) DECIMAL EXTERNAL(9,2)
                    NULLIF( 96)='?'
)

    To use this LOAD statement, type the changes you want to make,
    and delete the lines you do not need.
*/
L2 = WHEREAMI()
/*****
/* TRACE ?R                                           */
/*****
Address ISPEXEC
"ISREDIT MACRO (ARGS) NOPROCESS"
If ARGS = "" Then
Do
    Do I = L1+2 To L2-2;Say SourceLine(I);End
    Exit (20)
End
Parse Upper Var Args Table "(" Parms
Parms = Translate(Parms," ","")
Type = "SELECT" /* Default */
SSID = "" /* Default */
"VGET (DSNEOV01)"
If RC = 0 Then SSID = DSNEOV01
If (Parms <> "") Then
Do Until(Parms = "")
Parse Var Parms Var "=" Value Parms
    If Var = "T" | Var = "TYPE" Then Type = Value
    Else
    If Var = "S" | Var = "SSID" Then SSID = Value
    Else
    Exit (20)
End
"CONTROL ERRORS RETURN"
"ISREDIT (LEFTBND,RIGHTBND) = BOUNDS"
"ISREDIT (LRECL) = DATA_WIDTH" /*LRECL*/
BndSize = RightBnd - LeftBnd + 1
If BndSize > 72 Then BndSize = 72
"ISREDIT PROCESS DEST"
Select
    When rc = 0 Then
        'ISREDIT (ZDEST) = LINENUM .ZDEST'
    When rc <= 8 Then /* No A or B entered */
        Do
            zedsmsg = 'Enter "A"/"B" line cmd'
            zedlmsg = 'DRAW requires an "A" or "B" line command'
            'SETMSG MSG(ISRZ001)'
            Exit 12
        End
    When rc < 20 Then /* Conflicting line commands - edit sets message */
        Exit 12
    When rc = 20 Then
        zdest = 0
    Otherwise
        Exit 12
End
SQLTYPE. = "UNKNOWN TYPE"
VCHTYPE = 448; SQLTYPES.VCHTYPE = 'VARCHAR'
CHTYPE = 452; SQLTYPES.CHTYPE = 'CHAR'
LVCHTYPE = 456; SQLTYPES.LVCHTYPE = 'VARCHAR'
VGRTP = 464; SQLTYPES.VGRTP = 'VARGRAPHIC'

```

```

GRITYP      = 468; SQLTYPES.GRTYP      = 'GRAPHIC'
LVGRITYP    = 472; SQLTYPES.LVGRITYP   = 'VARGRAPHIC'
FLOTYP      = 480; SQLTYPES.FLOTYP     = 'FLOAT'
DCTYP       = 484; SQLTYPES.DCTYP      = 'DECIMAL'
INTYP       = 496; SQLTYPES.INTYP       = 'INTEGER'
SMTYP       = 500; SQLTYPES.SMTYP       = 'SMALLINT'
DATYP       = 384; SQLTYPES.DATYP       = 'DATE'
TITYP       = 388; SQLTYPES.TITYP       = 'TIME'
TSTYP       = 392; SQLTYPES.TSTYP       = 'TIMESTAMP'
Address TSO "SUBCOM DSNREXX"             /* HOST CMD ENV AVAILABLE? */
IF RC THEN                               /* NO, LET'S MAKE ONE */
  S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX') /* ADD HOST CMD ENV */
Address DSNREXX "CONNECT" SSID
If SQLCODE ^= 0 Then Call SQLCA
Address DSNREXX "EXECSQL DESCRIBE TABLE :TABLE INTO :SQLDA"
If SQLCODE ^= 0 Then Call SQLCA
Address DSNREXX "EXECSQL COMMIT"
Address DSNREXX "DISCONNECT"
If SQLCODE ^= 0 Then Call SQLCA
Select
  When (Left(Type,1) = "S") Then
    Call DrawSelect
  When (Left(Type,1) = "I") Then
    Call DrawInsert
  When (Left(Type,1) = "U") Then
    Call DrawUpdate
  When (Left(Type,1) = "L") Then
    Call DrawLoad
  Otherwise EXIT (20)
End
Do I = LINE.0 To 1 By -1
  LINE = COPIES(" ",LEFTBND-1)||LINE.I
  'ISREDIT LINE_AFTER 'zdest' = DATALINE (Line)'
End
line1 = zdest + 1
'ISREDIT CURSOR = 'line1 0
Exit

/*****
WHEREAMI;; RETURN SIGL
*****/
/* Draw SELECT */
/*****
DrawSelect:
  Line.0 = 0
  Line = "SELECT"
  Do I = 1 To SQLDA.SQLD
    If I > 1 Then Line = Line ','
    ColName = "'SQLDA.I.SQLNAME'"
    Null = SQLDA.I.SQLTYPE//2
    If Length(Line)+Length(ColName)+LENGTH(" ,") > BndSize THEN
      Do
        L = Line.0 + 1; Line.0 = L
        Line.L = Line
        Line = "      "
      End
      Line = Line ColName
    End I
    If Line ^= "" Then
      Do
        L = Line.0 + 1; Line.0 = L
        Line.L = Line
        Line = "      "
      End
      L = Line.0 + 1; Line.0 = L
      Line.L = "FROM" TABLE
    Return
*****/

```



```

/* Draw INSERT                                                                    */
/*****                                                                    */
DrawInsert:
  Line.0 = 0
  Line = "INSERT INTO" TABLE "("
  Do I = 1 To SQLDA.SQLD
    If I > 1 Then Line = Line ','
    ColName = '''SQLDA.I.SQLNAME'''
    If Length(Line)+Length(ColName) > BndSize THEN
      Do
        L = Line.0 + 1; Line.0 = L
        Line.L = Line
        Line = "      "
      End
      Line = Line ColName
      If I = SQLDA.SQLD Then Line = Line ')'
    End I
    If Line ^= "" Then
      Do
        L = Line.0 + 1; Line.0 = L
        Line.L = Line
        Line = "      "
      End
      L = Line.0 + 1; Line.0 = L
      Line.L = "  VALUES ("
      L = Line.0 + 1; Line.0 = L
      Line.L = ,
      "-- ENTER VALUES BELOW          COLUMN NAME          DATA TYPE"
      Do I = 1 To SQLDA.SQLD
        If SQLDA.SQLD > 1 & I < SQLDA.SQLD Then
          Line = "          , --"
        Else
          Line = "          ) --"
        End If
        Line = Line Left(SQLDA.I.SQLNAME,18)
        Type = SQLDA.I.SQLTYPE
        Null = Type//2
        If Null Then Type = Type - 1
        Len = SQLDA.I.SQLEN
        Prcsn = SQLDA.I.SQLEN.SQLPRECISION
        Scale = SQLDA.I.SQLEN.SQLSCALE
        Select
          When (Type = CHTYPE ,
                Type = VCHTYPE ,
                Type = LVCHTYPE ,
                Type = GRTYP ,
                Type = VGRTYP ,
                Type = LVGRTYP ) THEN
            Type = SQLTYPES.Type("STRIP(LEN)")
          When (Type = FLOTYPE ) THEN
            Type = SQLTYPES.Type("STRIP((LEN*4)-11) ")
          When (Type = DCTYPE ) THEN
            Type = SQLTYPES.Type("STRIP(PRCN) ", "STRIP(SCALE)")
          Otherwise
            Type = SQLTYPES.Type
        End
        Line = Line Type
        If Null = 0 Then
          Line = Line "NOT NULL"
          L = Line.0 + 1; Line.0 = L
          Line.L = Line
        End I
      End I
      Return
/*****                                                                    */
/* Draw UPDATE                                                                    */
/*****                                                                    */
DrawUpdate:

```

```

Line.0 = 1
Line.1 = "UPDATE" TABLE "SET"
L = Line.0 + 1; Line.0 = L
Line.L = ,
"-- COLUMN NAME          ENTER VALUES BELOW      DATA TYPE"
Do I = 1 To SQLDA.SQLD
  If I = 1 Then
    Line = " "
  Else
    Line = " ,"
  Line = Line Left('"'SQLDA.I.SQLNAME"'=',21)
  Line = Line Left(" ",20)
  Type = SQLDA.I.SQLTYPE
  Null = Type//2
  If Null Then Type = Type - 1
  Len = SQLDA.I.SQLEN
  Prcsn = SQLDA.I.SQLEN.SQLPRECISION
  Scale = SQLDA.I.SQLEN.SQLSCALE
  Select
  When (Type = CHTYPE ,
      |Type = VCHTYPE ,
      |Type = LVCHTYPE ,
      |Type = GRTYP ,
      |Type = VGRTP ,
      |Type = LVGRTP ) THEN
    Type = SQLTYPES.Type("STRIP(LEN)")
  When (Type = FLOTYPE ) THEN
    Type = SQLTYPES.Type("STRIP((LEN*4)-11) ")
  When (Type = DCTYPE ) THEN
    Type = SQLTYPES.Type("STRIP(PRCSN) ", "STRIP(SCALE)")
  Otherwise
    Type = SQLTYPES.Type
  End
  Line = Line "--" Type
  If Null = 0 Then
    Line = Line "NOT NULL"
  L = Line.0 + 1; Line.0 = L
  Line.L = Line
End I
L = Line.0 + 1; Line.0 = L
Line.L = "WHERE"
Return

/*****
/* Draw LOAD */
*****/
DrawLoad:
Line.0 = 1
Line.1 = "LOAD DATA INDDN SYSREC INTO TABLE" TABLE
Position = 1
Do I = 1 To SQLDA.SQLD
  If I = 1 Then
    Line = " ("
  Else
    Line = " ,"
  Line = Line Left('"'SQLDA.I.SQLNAME"'',20)
  Line = Line "POSITION("RIGHT(POSITION,5)")"
  Type = SQLDA.I.SQLTYPE
  Null = Type//2
  If Null Then Type = Type - 1
  Len = SQLDA.I.SQLEN
  Prcsn = SQLDA.I.SQLEN.SQLPRECISION
  Scale = SQLDA.I.SQLEN.SQLSCALE
  Select
  When (Type = CHTYPE ,
      |Type = GRTYP ) THEN
    Type = SQLTYPES.Type("STRIP(LEN)")
  When (Type = FLOTYPE ) THEN

```

```

        Type = SQLTYPES.Type("STRIP((LEN*4)-11) ")
When (Type = DCTYPE ) THEN
Do
    Type = SQLTYPES.Type "EXTERNAL"
    Type = Type("STRIP(PRCSN)","STRIP(SCALE)")
    Len = (PRCSN+2)%2
End
When (Type = DATYPE ,
      Type = TITYPE ,
      Type = TSTYPE ) THEN
    Type = SQLTYPES.Type "EXTERNAL"
Otherwise
    Type = SQLTYPES.Type
End
If (Type = GRTYP ,
    Type = VGRYP ,
    Type = LVGRYP ) THEN
    Len = Len * 2
If (Type = VCHTYPE ,
    Type = LVCHTYPE ,
    Type = VGRYP ,
    Type = LVGRYP ) THEN
    Len = Len + 2
Line = Line Type
L = Line.0 + 1; Line.0 = L

Line.L = Line
If Null = 1 Then
Do
    Line = " "
    Line = Line Left(' ',20)
    Line = Line " NULLIF("RIGHT(POSITION,5)")=?'"
    L = Line.0 + 1; Line.0 = L
    Line.L = Line
End
Position = Position + Len + 1
End I
L = Line.0 + 1; Line.0 = L
Line.L = " )"
Return
/*****
/* Display SQLCA */
/*****
SQLCA:
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLSTATE="SQLSTATE"'
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLWARN ="SQLWARN.0",',
    SQLWARN.1",",
    SQLWARN.2",",
    SQLWARN.3",",
    SQLWARN.4",",
    SQLWARN.5",",
    SQLWARN.6",",
    SQLWARN.7",",
    SQLWARN.8",",
    SQLWARN.9",",
    SQLWARN.10"'
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLERRD ="SQLERRD.1",',
    SQLERRD.2",",
    SQLERRD.3",",
    SQLERRD.4",",
    SQLERRD.5",",
    SQLERRD.6"'
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLERRP ="SQLERRP"'
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLERRMC ="SQLERRMC"'
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLCODE ="SQLCODE"'
Exit 20

```

Example of how an indicator variable is used in a REXX program

The way that you use indicator variables for input host variables in REXX programs is slightly different than the way that you use indicator variables in other languages. When you want to pass a null value to a DB2 column, in addition to putting a negative value in an indicator variable, you also need to put a valid value in the corresponding host variable.

For example, the following statements set a value in the WORKDEPT column in table EMP to null:

```
SQLSTMT="UPDATE EMP" ,
  "SET WORKDEPT = ?"
HVWORKDEPT='000'
INDWORKDEPT=-1
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING :HVWORKDEPT :INDWORKDEPT"
```

In the following program, the phone number for employee Haas is selected into variable HVPhone. After the SELECT statement executes, if no phone number for employee Haas is found, indicator variable INDPhone contains -1.

```
'SUBCOM DSNREXX'
IF RC THEN ,
  S_RC = RXSUBCOM('ADD', 'DSNREXX', 'DSNREXX')
ADDRESS DSNREXX
'CONNECT' 'DSN'
SQLSTMT = ,
  "SELECT PHONENO FROM DSN8A10.EMP WHERE LASTNAME='HAAS'"
"EXECSQL DECLARE C1 CURSOR FOR S1"
"EXECSQL PREPARE S1 FROM :SQLSTMT"
Say "SQLCODE from PREPARE is "SQLCODE
"EXECSQL OPEN C1"
Say "SQLCODE from OPEN is "SQLCODE
"EXECSQL FETCH C1 INTO :HVPhone :INDPhone"
Say "SQLCODE from FETCH is "SQLCODE
If INDPhone < 0 Then ,
  Say 'Phone number for Haas is null.'
"EXECSQL CLOSE C1"
Say "SQLCODE from CLOSE is "SQLCODE
S_RC = RXSUBCOM('DELETE', 'DSNREXX', 'DSNREXX')
```

Chapter 10. Creating and modifying DB2 objects

Your application program can create and manipulate DB2 objects, such as tables, views, triggers, distinct types, user-defined functions, and stored procedures. You must have the appropriate authorizations to create such objects.

Creating tables

Creating a table provides a logical place to store related data on a DB2 subsystem.

To create a table, use a CREATE TABLE statement that includes the following elements:

- The name of the table
- A list of the columns that make up the table. For each column, specify the following information:
 - The column's name (for example, SERIAL).
 - The data type and length attribute (for example, CHAR(8)).
 - Optionally, a default value.
 - Optionally, a referential constraint or check constraint.

Separate each column description from the next with a comma, and enclose the entire list of column descriptions in parentheses.

Example: The following SQL statement creates a table named PRODUCT:

```
CREATE TABLE PRODUCT
(SERIAL      CHAR(8)      NOT NULL,
DESCRIPTION  VARCHAR(60)  DEFAULT,
MFGCOST     DECIMAL(8,2),
MFGDEPT     CHAR(3),
MARKUP      SMALLINT,
SALESDEPT   CHAR(3),
CURDATE     DATE         DEFAULT);
```

For more information about referential constraints, see “Referential constraints” on page 471

For more information about check constraints, see “Check constraints” on page 469.

Identifying column defaults and constraining column inputs:

If you want to constrain the input or identify the default of a column, you can use the following values:

- NOT NULL, when the column cannot contain null values.
- UNIQUE, when the value for each row must be unique, and the column cannot contain null values.
- DEFAULT, when the column has one of the following DB2-assigned defaults:
 - For numeric columns, 0 (zero) is the default value.
 - For character or graphic fixed-length strings, blank is the default value.
 - For binary fixed-length strings, a set of hexadecimal zeros is the default value.

- For variable-length strings, including LOB strings, the empty string (a string of zero-length) is the default value.
- For datetime columns, the current value of the associated special register is the default value.
- **DEFAULT *value***, when you want to identify one of the following values as the default value:
 - A constant
 - NULL
 - SESSION_USER, which specifies the value of the SESSION_USER special register at the time when a default value is needed for the column
 - CURRENT SQLID, which specifies the value of the CURRENT SQLID special register at the time when a default value is needed for the column
 - The name of a cast function that casts a default value (of a built-in data type) to the distinct type of a column

Related reference:

 [CREATE TABLE \(DB2 SQL\)](#)

Related information:

 [Implementing DB2 tables \(DB2 Administration Guide\)](#)

Data types

When you create a DB2 table, you define each column to have a specific data type. The data type of a column determines what you can and cannot do with the column.

When you perform operations on columns, the data must be compatible with the data type of the referenced column. For example, you cannot insert character data, such as a last name, into a column whose data type is numeric. Similarly, you cannot compare columns that contain incompatible data types.

The data type for a column can be a distinct type, which is a user-defined data type, or a DB2 built-in data type. As shown in the following figure, DB2 built-in data types have four general categories: datetime, string, numeric, and row identifier (ROWID).

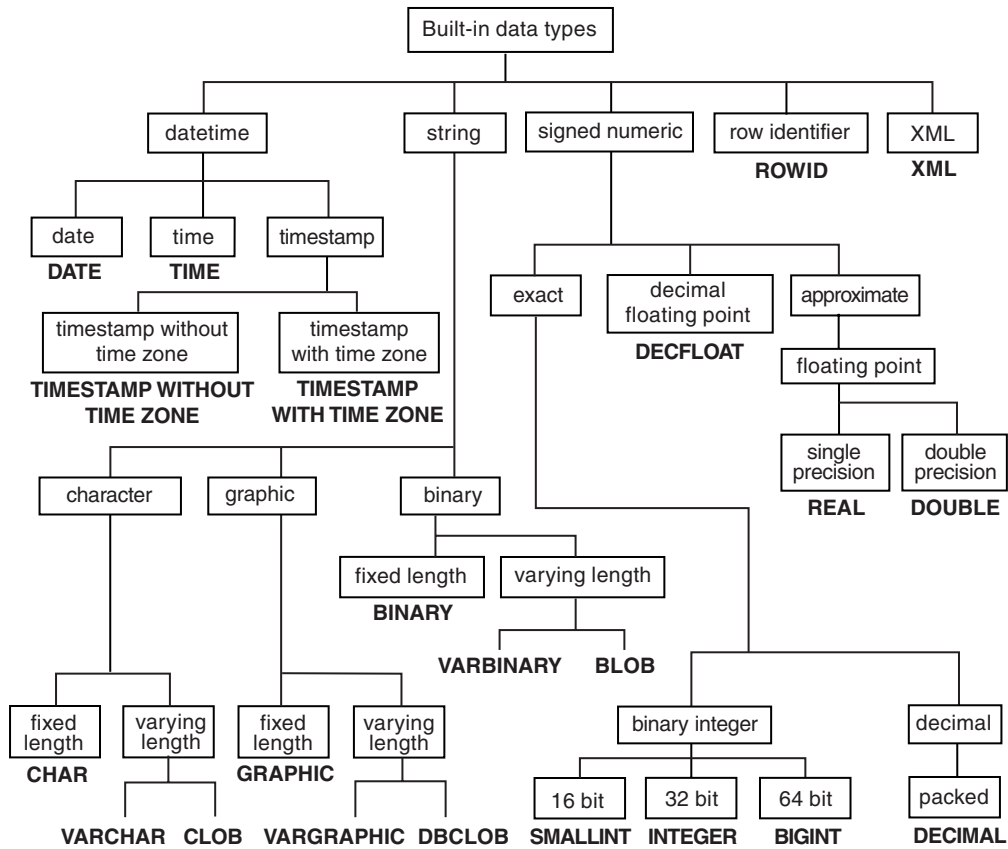


Figure 23. DB2 built-in data types

The following table shows whether operands of any two data types are compatible, Y (Yes), or incompatible, N (No). Notes are indicated either as a superscript number next to Y or N or as a value in the column of the table.

Table 77. Supported casts between built-in data types

Cast from data type –	To data type ¹																			
	S M A L L I N T	I N T E G E R	B I G I N T	D E C I M A L	D E C I M A L	R E A L	D O U B L E	V A R C H A R	C H A R	C H A R	O B J E C T	G R A P H I C	V A R C H A R	D B I N A R Y	V A R B I N A R Y	B L O B	D E C I M A L	T I M E	T I M E S T A M P	T I M E S T A M P W I T H O U T T I M E Z O N E
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y											
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y											
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y											
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y											
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y											
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y											
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y											
CHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
VARCHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CLOB							Y	Y	Y	Y	Y	Y	Y	Y	Y					
GRAPHIC	Y	Y	Y	Y	Y	Y	Y	Y ²	Y ²	Y ²	Y	Y	Y	Y	Y	Y	Y ³	Y ³	Y ³	Y ³
VARGRAPHIC	Y	Y	Y	Y	Y	Y	Y	Y ²	Y ²	Y ²	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y ³
DBCLOB							Y ²	Y ²	Y ²	Y	Y	Y	Y	Y	Y					
BINARY													Y	Y	Y					
VARBINARY													Y	Y	Y					
BLOB													Y	Y	Y					
DATE							Y	Y								Y		Y		
TIME							Y	Y									Y			
TIMESTAMP WITHOUT TIME ZONE							Y	Y								Y	Y	Y	Y	

Define only one ROWID column, even if the table is to have multiple LOB columns. If you do not create a ROWID column before you define a LOB column, DB2 creates an implicitly hidden ROWID column and appends it as the last column of the table.

If you add a ROWID column after you add a LOB column, the table has two ROWID columns: the implicitly-created, hidden, column and the explicitly-created column. In this case, DB2 ensures that the values of the two ROWID columns are always identical.

If DB2 implicitly creates the table space for this table or CURRENT RULES is set to STD, DB2 creates the necessary auxiliary objects for you and you can skip steps 2 and 3.

2. If you explicitly created the table space for this table and the CURRENT RULES special register is not set to STD, create a LOB table space and auxiliary table by using the CREATE LOB TABLESPACE and CREATE AUXILIARY TABLE statements.
 - If your base table is nonpartitioned, create one LOB table space and for each column create one auxiliary table.
 - If your base table is partitioned, create one LOB table space for each partition and one auxiliary table for each column. For example, if your base table has three partitions, you must create three LOB table spaces and three auxiliary tables for each LOB column.
3. If you explicitly created the table space for this table and the CURRENT RULES special register is not set to STD, create one index for each auxiliary table by using the CREATE INDEX statement.
4. Insert the LOB data into DB2 by using one of the following techniques:
 - If the total length of a LOB column and the base table row is less than 32 KB, use the LOAD utility and specify the base table.
 - Otherwise, use INSERT, UPDATE, or MERGE statements and specify the base table. If you use the INSERT statement, ensure that your application has enough storage available to hold the entire value that is to be put into the LOB column.

Example: Adding a CLOB column: Suppose that you want to add a resume for each employee to the employee table. The employee resumes are no more than 5 MB in size. Because the employee resumes contain single-byte characters, you can define the resumes to DB2 as CLOBs. You therefore need to add a column of data type CLOB with a length of 5 MB to the employee table. If you want to define a ROWID column explicitly, you must define it before you define the CLOB column.

First, execute an ALTER TABLE statement to add the ROWID column, and then execute another ALTER TABLE statement to add the CLOB column. The following statements create these columns:

```
ALTER TABLE EMP
  ADD ROW_ID ROWID NOT NULL GENERATED ALWAYS;
COMMIT;
ALTER TABLE EMP
  ADD EMP_RESUME CLOB(5M);
COMMIT;
```

If you explicitly created the table space for this table and the CURRENT RULES special register is not set to STD, you then need to define a LOB table space and an auxiliary table to hold the employee resumes. You also need to define an index on the auxiliary table. You must define the LOB table space in the same database as the associated base table. The following statements create these objects:

```

CREATE LOB TABLESPACE RESUMETS
  IN DSN8D10A
  LOG NO
  COMMIT;
CREATE AUXILIARY TABLE EMP_RESUME_TAB
  IN DSN8D10A.RESUMETS
  STORES DSN8A10.EMP
  COLUMN EMP_RESUME;
CREATE UNIQUE INDEX XEMP_RESUME
  ON EMP_RESUME_TAB;
COMMIT;

```

You can then load your employee resumes into DB2. In your application, you can define a host variable to hold the resume, copy the resume data from a file into the host variable, and then execute an UPDATE statement to copy the data into DB2. Although the LOB data is stored in the auxiliary table, your UPDATE statement specifies the name of the base table. The following code declares a host variable to store the resume in the C language:

```
SQL TYPE is CLOB (5M) resumedata;
```

The following UPDATE statement copies the data into DB2:

```

UPDATE EMP SET EMP_RESUME=:resumedata
  WHERE EMPNO=:employeeenum;

```

In this statement, `employeeenum` is a host variable that identifies the employee who is associated with a resume.

Large objects (LOBs)

The term *large object* and the acronym LOB refer to DB2 objects that you can use to store large amounts of data. A LOB is a varying-length character string that can contain up to 2 GB - 1 of data.

The three LOB data types are:

- *Binary large object (BLOB)*
Use a BLOB to store binary data such as pictures, voice, and mixed media.
- *Character large object (CLOB)*
Use a CLOB to store SBCS or mixed character data, such as documents.
- *Double-byte character large object (DBCLOB)*
Use a DBCLOB to store data that consists of only DBCS data.

You can use DB2 to store LOB data, but this data is stored differently than other kinds of data.

Although a table can have a LOB column, the actual LOB data is stored in a another table, which called the auxiliary table. This auxiliary table exists in a separate table space called a LOB table space. One auxiliary table must exist for each LOB column. The table with the LOB column is called the base table. The base table has a ROWID column that DB2 uses to locate the data in the auxiliary table. The auxiliary table must have exactly one index.

Implicitly hidden ROWID columns

If you do not create a ROWID column before you define a LOB column, DB2 creates an implicitly hidden ROWID column for you. This column is accessible only if you reference the column directly. The column is not included in the results of SELECT * statements or DESCRIBE statements.

DB2 assigns the `GENERATED ALWAYS` attribute and the name `DB2_GENERATED_ROWID_FOR_LOBS nm` to an implicitly hidden ROWID column. DB2 appends the identifier nm only if the column name already exists in the table. If so, DB2 appends 00 and increments by 1 until the name is unique within the row.

Related reference:

 ALTER TABLE (DB2 SQL)

 ALTER VIEW (DB2 SQL)

 CREATE TABLE (DB2 SQL)

 select-clause (DB2 SQL)

Identity columns

An identity column contains a unique numeric value for each row in the table. DB2 can automatically generate sequential numeric values for this column as rows are inserted into the table. Thus, identity columns are ideal for primary key values, such as employee numbers or product numbers.

Using identity columns as keys

If you define a column with the `AS IDENTITY` attribute, and with the `GENERATED ALWAYS` and `NO CYCLE` attributes, DB2 automatically generates a monotonically increasing or decreasing sequential number for the value of that column when a new row is inserted into the table. However, for DB2 to guarantee that the values of the identity column are unique, you should define a unique index on that column.

You can use identity columns for primary keys that are typically unique sequential numbers, for example, order numbers or employee numbers. By doing so, you can avoid the concurrency problems that can result when an application generates its own unique counter outside the database.

Recommendation: Set the values of the foreign keys in the dependent tables after loading the parent table. If you use an identity column as a parent key in a referential integrity structure, loading data into that structure could be quite complicated. The values for the identity column are not known until the table is loaded because the column is defined as `GENERATED ALWAYS`.

You might have gaps in identity column values for the following reasons:

- If other applications are inserting values into the same identity column
- If DB2 terminates abnormally before it assigns all the cached values
- If your application rolls back a transaction that inserts identity values

Defining an identity column

You can define an identity column as either `GENERATED BY DEFAULT` or `GENERATED ALWAYS`:

- If you define the column as `GENERATED BY DEFAULT`, you can insert a value, and DB2 provides a default value if you do not supply one.
- If you define the column as `GENERATED ALWAYS`, DB2 always generates a value for the column, and you cannot insert data into that column. If you want the values to be unique, you must define the identity column with `GENERATED ALWAYS` and `NO CYCLE` and define a unique index on that column.

The values that DB2 generates for an identity column depend on how the column is defined. The `START WITH` option determines the first value that DB2 generates. The values advance by the `INCREMENT BY` value in ascending or descending order.

The `MINVALUE` and `MAXVALUE` options determine the minimum and maximum values that DB2 generates. The `CYCLE` or `NO CYCLE` option determines whether DB2 wraps values when it has generated all values between the `START WITH` value and `MAXVALUE` if the values are ascending, or between the `START WITH` value and `MINVALUE` if the values are descending.

Example: Using `GENERATED ALWAYS` and `CYCLE`

Suppose that table `T1` is defined with `GENERATED ALWAYS` and `CYCLE`:

```
CREATE TABLE T1
  (CHARCOL1 CHAR(1),
   IDENTCOL1 SMALLINT GENERATED ALWAYS AS IDENTITY
   (START WITH -1,
    INCREMENT BY 1,
    CYCLE,
    MINVALUE -3,
    MAXVALUE 3));
```

Now suppose that you execute the following `INSERT` statement eight times:

```
INSERT INTO T1 (CHARCOL1) VALUES ('A');
```

When DB2 generates values for `IDENTCOL1`, it starts with `-1` and increments by `1` until it reaches the `MAXVALUE` of `3` on the fifth `INSERT`. To generate the value for the sixth `INSERT`, DB2 cycles back to `MINVALUE`, which is `-3`. `T1` looks like this after the eight `INSERT`s are executed:

CHARCOL1	IDENTCOL1
A	-1
A	0
A	1
A	2
A	3
A	-3
A	-2
A	-1

The value of `IDENTCOL1` for the eighth `INSERT` repeats the value of `IDENTCOL1` for the first `INSERT`.

Identity columns as primary keys

The `SELECT` from `INSERT` statement enables you to insert a row into a parent table with its primary key defined as a DB2-generated identity column, and retrieve the value of the primary or parent key. You can then use this generated value as a foreign key in a dependent table.

In addition, you can use the `IDENTITY_VAL_LOCAL` function to return the most recently assigned value for an identity column.

Example: Using `SELECT` from `INSERT`

Suppose that an `EMPLOYEE` table and a `DEPARTMENT` table are defined in the following way:

```

CREATE TABLE EMPLOYEE
  (EMPNO      INTEGER GENERATED ALWAYS AS IDENTITY
   PRIMARY KEY NOT NULL,
   NAME       CHAR(30) NOT NULL,
   SALARY     DECIMAL(7,2) NOT NULL,
   WORKDEPT  SMALLINT);

CREATE TABLE DEPARTMENT
  (DEPTNO    SMALLINT NOT NULL PRIMARY KEY,
   DEPTNAME  VARCHAR(30),
   MGRNO     INTEGER NOT NULL,
   CONSTRAINT REF_EMPNO FOREIGN KEY (MGRNO)
   REFERENCES EMPLOYEE (EMPNO) ON DELETE RESTRICT);

ALTER TABLE EMPLOYEE ADD
  CONSTRAINT REF_DEPTNO FOREIGN KEY (WORKDEPT)
  REFERENCES DEPARTMENT (DEPTNO) ON DELETE SET NULL;

```

When you insert a new employee into the EMPLOYEE table, to retrieve the value for the EMPNO column, you can use the following SELECT from INSERT statement:

```

EXEC SQL
  SELECT EMPNO INTO :hv_empno
  FROM FINAL TABLE (INSERT INTO EMPLOYEE (NAME, SALARY, WORKDEPT)
  VALUES ('New Employee', 75000.00, 11));

```

The SELECT statement returns the DB2-generated identity value for the EMPNO column in the host variable :hv_empno.

You can then use the value in :hv_empno to update the MGRNO column in the DEPARTMENT table with the new employee as the department manager:

```

EXEC SQL
  UPDATE DEPARTMENT
  SET MGRNO = :hv_empno
  WHERE DEPTNO = 11;

```

Related concepts:

“Rules for inserting data into an identity column” on page 658

Related tasks:

“Selecting values while inserting data” on page 663

Related reference:

 [IDENTITY_VAL_LOCAL \(DB2 SQL\)](#)

Creating tables for data integrity

To ensure that only valid data is added to your tables, you can use constraints, triggers, and unique indexes. For example, you might need to ensure that all items in your inventory table have valid item numbers and to prevent items without valid item numbers from being added.

 **Introductory concepts:**

DB2 and data integrity (Introduction to DB2 for z/OS)

Creation of relationships with referential constraints (Introduction to DB2 for z/OS)

Related tasks:

 [Altering a table for referential integrity \(DB2 Administration Guide\)](#)

- Creating indexes to improve referential integrity performance for foreign keys (DB2 Performance)
- Creating tables for data integrity (DB2 Application programming and SQL)
- Using referential integrity for data consistency (Managing Security)

Ways to maintain data integrity

When you add or modify data in a DB2 table, you need to ensure that the data is valid. Two techniques that you can use to ensure valid data are constraints and triggers.

Constraints are rules that limit the values that you can insert, delete, or update in a table. There are two types of constraints:

- Check constraints determine the values that a column can contain. Check constraints are discussed in “Check constraints.”
- Referential constraints preserve relationships between tables. Referential constraints are discussed in “Referential constraints” on page 471. A specific type of referential constraints, the informational referential constraint, is discussed in “Informational referential constraints” on page 473.

Triggers are a series of actions that are invoked when a table is updated. Triggers are discussed in “Creating triggers” on page 493.

Check constraints:

A *check constraint* is a rule that specifies the values that are allowed in one or more columns of every row of a base table. For example, you can define a check constraint to ensure that all values in a column that contains ages are positive numbers.

Check constraints designate the values that specific columns of a base table can contain, providing you a method of controlling the integrity of data entered into tables. You can create tables with check constraints using the CREATE TABLE statement, or you can add the constraints with the ALTER TABLE statement. However, if the check integrity is compromised or cannot be guaranteed for a table, the table space or partition that contains the table is placed in a check pending state. Check integrity is the condition that exists when each row of a table conforms to the check constraints defined on that table.

For example, you might want to make sure that no salary can be below 15000 dollars. To do this, you can create the following check constraint:

```
CREATE TABLE EMP_SAL
(ID          INTEGER      NOT NULL,
 SALARY     INTEGER      CHECK (SALARY >= 15000));
```

Using check constraints makes your programming task easier, because you do not need to enforce those constraints within application programs or with a validation routine. Define check constraints on one or more columns in a table when that table is created or altered.

Check constraint considerations

The syntax of a check constraint is checked when the constraint is defined, but the meaning of the constraint is not checked. The following examples show mistakes that are not caught. Column C1 is defined as INTEGER NOT NULL.

Allowable but mistaken check constraints:

- A self-contradictory check constraint:
`CHECK (C1 > 5 AND C1 < 2)`
- Two check constraints that contradict each other:
`CHECK (C1 > 5)`
`CHECK (C1 < 2)`
- Two check constraints, one of which is redundant:
`CHECK (C1 > 0)`
`CHECK (C1 >= 1)`
- A check constraint that contradicts the column definition:
`CHECK (C1 IS NULL)`
- A check constraint that repeats the column definition:
`CHECK (C1 IS NOT NULL)`

A check constraint is not checked for consistency with other types of constraints. For example, a column in a dependent table can have a referential constraint with a delete rule of SET NULL. You can also define a check constraint that prohibits nulls in the column. As a result, an attempt to delete a parent row fails, because setting the dependent row to null violates the check constraint.

Similarly, a check constraint is not checked for consistency with a validation routine, which is applied to a table before a check constraint. If the routine requires a column to be greater than or equal to 10 and a check constraint requires the same column to be less than 10, table inserts are not possible. Plans and packages do not need to be rebound after check constraints are defined on or removed from a table.

When check constraints are enforced

After check constraints are defined on a table, any change must satisfy those constraints if it is made by:

- The LOAD utility with the option ENFORCE CONSTRAINT
- An SQL insert operation
- An SQL update operation

A row satisfies a check constraint if its condition evaluates either to true or to unknown. A condition can evaluate to unknown for a row if one of the named columns contains the null value for that row.

Any constraint defined on columns of a base table applies to the views defined on that base table.

When you use ALTER TABLE to add a check constraint to already populated tables, the enforcement of the check constraint is determined by the value of the CURRENT RULES special register as follows:

- If the value is STD, the check constraint is enforced immediately when it is defined. If a row does not conform, the check constraint is not added to the table and an error occurs.
- If the value is DB2, the check constraint is added to the table description but its enforcement is deferred. Because there might be rows in the table that violate the check constraint, the table is placed in CHECK-pending status.

CHECK-pending status:

To maintain data integrity DB2 enforces check constraints and referential constraints on data in a table. When these types of constraints are violated or might be violated, DB2 places the table space or partition that contains the table in CHECK-pending status.

Table check violations place a table space or partition in CHECK-pending status when any of these conditions exist:

- A check constraint is defined on a populated table using the ALTER TABLE statement, and the value of the CURRENT RULES special register is DB2.
- The LOAD utility is run with CONSTRAINTS NO, and check constraints are defined on the table.
- CHECK DATA is run on a table that contains violations of check constraints.
- A point-in-time RECOVER introduces violations of check constraints.

Referential constraints:

A *referential constraint* is a rule that specifies that the only valid values for a particular column are those values that exist in another specified table column. For example, a referential constraint can ensure that all customer IDs in a transaction table exist in the ID column of a customer table.

A table can serve as the “master list” of all occurrences of an entity. In the sample application, the employee table serves that purpose for employees; the numbers that appear in that table are the only valid employee numbers. Likewise, the department table provides a master list of all valid department numbers; the project activity table provides a master list of activities performed for projects; and so on.

The following figure shows the relationships that exist among the tables in the sample application. Arrows point from parent tables to dependent tables.

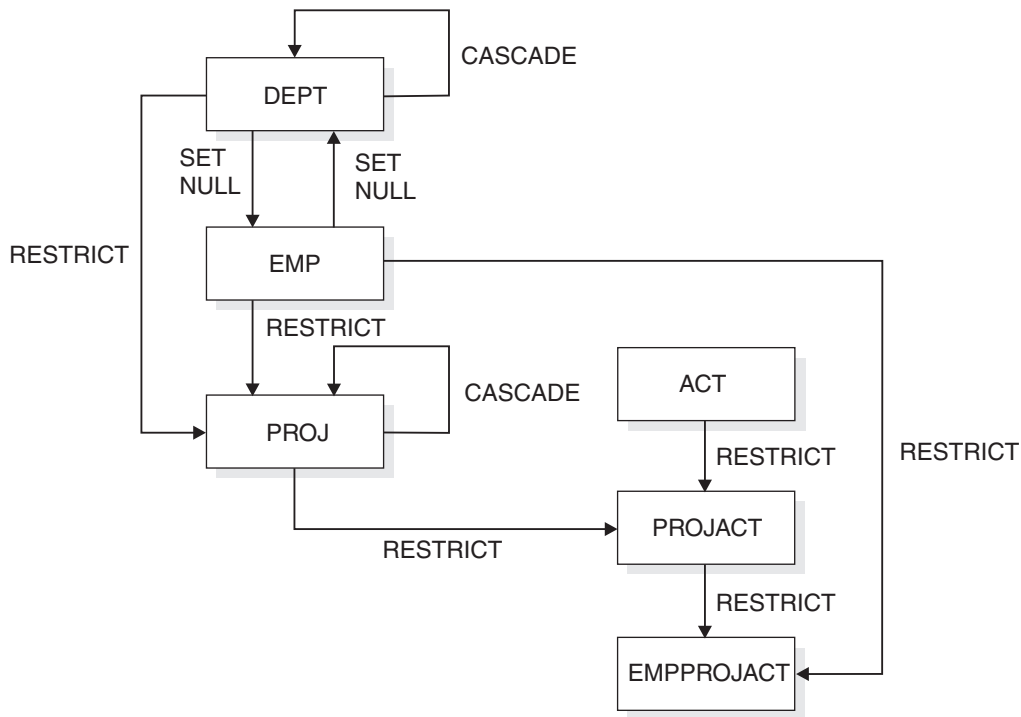


Figure 24. Relationships among tables in the sample application

When a table refers to an entity for which there is a master list, it should identify an occurrence of the entity that actually appears in the master list; otherwise, either the reference is invalid or the master list is incomplete. Referential constraints enforce the relationship between a table and a master list.

Restrictions on cycles of dependent tables:

A *cycle* is a set of two or more tables. The tables are ordered so that each is a dependent of the one before it, and the first is a dependent of the last. Every table in the cycle is a descendent of itself. DB2 restricts certain operations on cycles.

In the sample application, the employee and department tables are a cycle; each is a dependent of the other.

DB2 does not allow you to create a cycle in which a delete operation on a table involves that same table. Enforcing that principle creates rules about adding a foreign key to a table:

- In a cycle of two tables, neither delete rule can be CASCADE.
- In a cycle of more than two tables, two or more delete rules must not be CASCADE. For example, in a cycle with three tables, two of the delete rules must be other than CASCADE. This concept is illustrated in The following figure. The cycle on the left is valid because two or more of the delete rules are not CASCADE. The cycle on the right is invalid because it contains two cascading deletes.

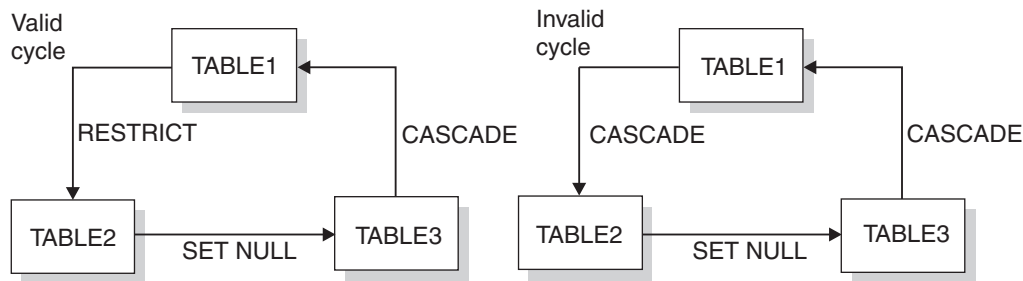


Figure 25. Valid and invalid delete cycles

Alternatively, a delete operation on a self-referencing table must involve the same table, and the delete rule there must be CASCADE or NO ACTION.

Recommendation: Avoid creating a cycle in which all the delete rules are RESTRICT and none of the foreign keys allows nulls. If you do this, no row of any of the tables can ever be deleted.

Referential constraints on tables with multilevel security with row-level granularity:

You cannot use referential constraints on a security label column, which is used for multilevel security with row-level granularity. However, you can use referential constraints on other columns in the row.

DB2 does not enforce multilevel security with row-level granularity when it is already enforcing referential constraints. Referential constraints are enforced when the following situations occur:

- An insert operation is applied to a dependent table.
- An update operation is applied to a foreign key of a dependent table, or to the parent key of a parent table.
- A delete operation is applied to a parent table. In addition to all referential constraints being enforced, the DB2 system enforces all delete rules for all dependent rows that are affected by the delete operation. If all referential constraints and delete rules are not satisfied, the delete operation will not succeed.
- The LOAD utility with the ENFORCE CONSTRAINTS option is run on a dependent table.
- The CHECK DATA utility is run.

Related concepts:

 [Multilevel security \(Managing Security\)](#)

Informational referential constraints:

An informational referential constraint is a referential constraint that DB2 does not enforce during normal operations. Use these constraints only when referential integrity can be enforced by another means, such as when retrieving data from other sources. These constraints might improve performance by enabling the query to qualify for automatic query rewrite.

DB2 ignores informational referential constraints during insert, update, and delete operations. Some utilities ignore these constraints; other utilities recognize them.


For example, CHECK DATA and LOAD ignore these constraints. QUIESCE TABLESPACESET recognizes these constraints by quiescing all table spaces related to the specified table space.

You should use this type of referential constraint only when an application process verifies the data in a referential integrity relationship. For example, when inserting a row in a dependent table, the application should verify that a foreign key exists as a primary or unique key in the parent table. To define an informational referential constraint, use the NOT ENFORCED option of the referential constraint definition in a CREATE TABLE or ALTER TABLE statement.

Informational referential constraints are often useful, especially in a data warehouse environment, for several reasons:

- To avoid the overhead of enforcement by DB2.
Typically, data in a data warehouse has been extracted and cleansed from other sources. Referential integrity might already be guaranteed. In this situation, enforcement by DB2 is unnecessary.
- To allow more queries to qualify for automatic query rewrite.
Automatic query rewrite is a process that examines a submitted query that references source tables and, if appropriate, rewrites the query so that it executes against a materialized query table that has been derived from those source tables. This process uses informational referential constraints to determine whether the query can use a materialized query table. Automatic query rewrite results in a significant reduction in query run time, especially for decision-support queries that operate over huge amounts of data.

Related tasks:

 [Using materialized query tables to improve SQL performance \(DB2 Performance\)](#)

Related reference:

 [CREATE TABLE \(DB2 SQL\)](#)

Defining a parent key and unique index

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

The primary key of a table, if one exists, uniquely identifies each occurrence of an entity in the table. The PRIMARY KEY clause of the CREATE TABLE or ALTER TABLE statements identifies the column or columns of the primary key. Each identified column must be defined as NOT NULL.

Another way to allow only unique values in a column is to specify the UNIQUE clause when you create or alter a table.

A table that is to be a parent of dependent tables must have a primary or a unique key; the foreign keys of the dependent tables refer to the primary or unique key. Otherwise, a primary key is optional. Consider defining a primary key if each row of your table does pertain to a unique occurrence of some entity. If you define a primary key, an index must be created (the *primary index*) on the same set of columns, in the same order as those columns. If you are defining referential constraints for DB2 to enforce, takes steps to maintain data integrity read before creating or altering any of the tables involved.

A table can have no more than one primary key. A primary key has the same restrictions as index keys:

- The key can include no more than 64 columns.
- You cannot specify a column name twice.
- The sum of the column length attributes cannot be greater than 2000.

You define a list of columns as the primary key of a table with the PRIMARY KEY clause in the CREATE TABLE statement.

To add a primary key to an existing table, use the PRIMARY KEY clause in an ALTER TABLE statement. In this case, a unique index must already exist.

Recommendations for defining primary keys:

Consider the following items when you plan for primary keys:

- The theoretical model of a relational database suggests that every table should have a primary key to uniquely identify the entities it describes. However, you must weigh that model against the potential cost of index maintenance overhead. DB2 does not require you to define a primary key for tables with no dependents.
- Choose a primary key whose values will not change over time. Choosing a primary key with persistent values enforces the good practice of having unique identifiers that remain the same for the lifetime of the entity occurrence.
- A primary key column should not have default values unless the primary key is a single TIMESTAMP column.
- Choose the minimum number of columns to ensure uniqueness of the primary key.
- A view that can be updated that is defined on a table with a primary key should include all columns of the key. Although this is necessary only if the view is used for inserts, the unique identification of rows can be useful if the view is used for updates, deletes, or selects.
- Drop a primary key later if you change your database or application using SQL.

Related concepts:

“Ways to maintain data integrity” on page 469

Related reference:

 ALTER TABLE (DB2 SQL)

 CREATE TABLE (DB2 SQL)

Parent key columns:

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

If every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the *parent key* of the table. To ensure that the parent key does not contain duplicate values, you must create a unique index on the column or columns that constitute the parent key. Defining the parent key is called entity integrity, because it requires each entity to have a unique key.

In some cases, using a timestamp as part of the key can be helpful, for example when a table does not have a “natural” unique key or if arrival sequence is the key.

Primary keys for some of the sample tables are:

Table Key Column

Employee table

EMPNO

Department table

DEPTNO

Project table

PROJNO

Table 78 shows part of the project table which has the primary key column, PROJNO.

Table 78. Part of the project table with the primary key column, PROJNO

PROJNO	PROJNAME	DEPTNO
MA2100	WELD LINE AUTOMATION	D01
MA2110	W L PROGRAMMING	D11

Table 79 shows part of the project activity table, which has a primary key that contains more than one column. The primary key is a *composite key*, which consists of the PRONNO, ACTNO, and ACSTDATE columns.

Table 79. Part of the Project activities table with a composite primary key

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3100	10	.50	1982-01-01	1982-07-01
AD3110	10	1.00	1982-01-01	1983-01-01
AD3111	60	.50	1982-03-15	1982-04-15

Defining a foreign key

Use foreign keys to enforce referential relationships between tables. A *foreign key* is a column or set of columns that references the parent key in the parent table.

The following prerequisites are met:

- The privilege set must include the ALTER or the REFERENCES privilege on the columns of the parent key.
- A unique index exists on the parent key columns of the parent table.

To define a foreign key, use one of the following approaches:

- Issue a CREATE TABLE statement and specify a FOREIGN KEY clause.
 1. Choose a constraint name for the relationship that is defined by a foreign key. If you do not choose a name, DB2 generates one from the name of the first column of the foreign key, in the same way that it generates the name of an implicitly created table space. For example, the names of the relationships in which the employee-to-project activity table is a dependent would, by default, be recorded (in column RELNAME of SYSIBM.SYSFOREIGNKEYS) as EMPNO and PROJNO.

The name is used in error messages, queries to the catalog, and DROP FOREIGN KEY statements. Hence, you might want to choose one if you are experimenting with your database design and have more than one foreign key that begins with the same column (otherwise DB2 generates the name).

2. Specify column names that identify the columns of the parent key. A foreign key can refer to either a unique or a primary key of the parent table. If the foreign key refers to a non-primary unique key, you must specify the column names of the key explicitly. If the column names of the key are not specified explicitly, the default is to refer to the column names of the primary key of the parent table.
- Issue an ALTER TABLE statement and specify the FOREIGN KEY clause. You can add a foreign key to an existing table; in fact, that is sometimes the only way to proceed. To make a table self-referencing, you must add a foreign key after creating it. When a foreign key is added to a populated table, the table space is put into CHECK-pending status.

The following example shows a CREATE TABLE statement that specifies constraint names REPAPA and REPAE for the foreign keys in the employee-to-project activity table.

```
CREATE TABLE DSN8A10.EMPPROJACT
  (EMPNO    CHAR(6)      NOT NULL,
   PROJNO   CHAR(6)      NOT NULL,
   ACTNO    SMALLINT     NOT NULL,
   CONSTRAINT REPAPA FOREIGN KEY (PROJNO, ACTNO)
     REFERENCES DSN8A10.PROJACT ON DELETE RESTRICT,
   CONSTRAINT REPAE FOREIGN KEY (EMPNO)
     REFERENCES DSN8A10.EMP ON DELETE RESTRICT)
IN DATABASE DSN8D10A;
```

If rows of the parent table are often deleted, it is best to create an index on the foreign key.

Related tasks:

 [Adding parent keys and foreign keys \(DB2 Administration Guide\)](#)

Related reference:

 [CREATE TABLE \(DB2 SQL\)](#)

 [ALTER TABLE \(DB2 SQL\)](#)

 [SYSIBM.SYSFOREIGNKEYS table \(DB2 SQL\)](#)

Related information:

Maintaining referential integrity when using data encryption

If you use encrypted data in a referential constraint, the primary key of the parent table and the foreign key of the dependent table must have the same encrypted value.

The encrypted value should be extracted from the parent table (the primary key) and used for the dependent table (the foreign key). You can do this in one of the following two ways:

- Use the FINAL TABLE clause on a SELECT from UPDATE, SELECT from INSERT, or SELECT from MERGE statement.
- Use the ENCRYPT_TDES function to encrypt the foreign key using the same password as the primary key. The encrypted value of the foreign key will be the same as the encrypted value of the primary key.

The SET ENCRYPTION PASSWORD statement sets the password that will be used for the ENCRYPT_TDES function.

Related reference:

[➡ ENCRYPT_TDES \(DB2 SQL\)](#)

[➡ ENCRYPTION PASSWORD \(DB2 SQL\)](#)

Creating work tables for the EMP and DEPT sample tables

Before testing SQL statements that insert, update, and delete rows in the DSN8A10.EMP and DSN8A10.DEPT sample tables, you should create duplicates of these tables. Create duplicates so that the original sample tables remain intact. These duplicate tables are called *work tables*.

This topic shows how to create the department and employee work tables and how to fill a work table with the contents of another table:

Each of these topics assumes that you logged on by using your own authorization ID. The authorization ID qualifies the name of each object that you create. For example, if your authorization ID is SMITH, and you create table YDEPT, the name of the table is SMITH.YDEPT. If you want to access table DSN8A10.DEPT, you must refer to it by its complete name. If you want to access your own table YDEPT, you need only to refer to it as YDEPT.

Use the following statements to create a new department table called YDEPT, modeled after the existing table, DSN8A10.DEPT, and an index for YDEPT:

```
CREATE TABLE YDEPT
  LIKE DSN8A10.DEPT;
CREATE UNIQUE INDEX YDEPTX
  ON YDEPT (DEPTNO);
```

If you want DEPTNO to be a primary key, as in the sample table, explicitly define the key. Use an ALTER TABLE statement, as in the following example:

```
ALTER TABLE YDEPT
  PRIMARY KEY(DEPTNO);
```

You can use an INSERT statement to copy the rows of the result table of a fullselect from one table to another. The following statement copies all of the rows from DSN8A10.DEPT to your own YDEPT work table:

```
INSERT INTO YDEPT
  SELECT *
  FROM DSN8A10.DEPT;
```

For information about using the INSERT statement, see “Inserting rows by using the INSERT statement” on page 655.

You can use the following statements to create a new employee table called YEMP:

```
CREATE TABLE YEMP
  (EMPNO      CHAR(6)          PRIMARY KEY NOT NULL,
   FIRSTNME  VARCHAR(12)     NOT NULL,
   MIDINIT   CHAR(1)         NOT NULL,
   LASTNAME  VARCHAR(15)     NOT NULL,
   WORKDEPT  CHAR(3)         REFERENCES YDEPT
                                ON DELETE SET NULL,
   PHONENO   CHAR(4)         UNIQUE NOT NULL,
   HIREDATE  DATE             ,
   JOB       CHAR(8)         ,
   EDLEVEL   SMALLINT        ,
```



```

SEX          CHAR(1)          ,
BIRTHDATE   DATE             ,
SALARY       DECIMAL(9, 2)    ,
BONUS        DECIMAL(9, 2)    ,
COMM         DECIMAL(9, 2)    );

```

This statement also creates a referential constraint between the foreign key in YEMP (WORKDEPT) and the primary key in YDEPT (DEPTNO). It also restricts all phone numbers to unique numbers.

If you want to change a table definition after you create it, use the ALTER TABLE statement with a RENAME clause. If you want to change a table name after you create it, use the RENAME statement.

You can change a table definition by using the ALTER TABLE statement only in certain ways. For example, you can add and drop constraints on columns in a table. You can also change the data type of a column within character data types, within numeric data types, and within graphic data types. You can add a column to a table. However, you cannot use the ALTER TABLE statement to drop a column from a table.

Related tasks:

[↗ Altering DB2 tables \(DB2 Administration Guide\)](#)

Related reference:

[↗ ALTER TABLE \(DB2 SQL\)](#)

[↗ RENAME \(DB2 SQL\)](#)

Creating created temporary tables

Use created temporary tables when you need to store data for only the life of an application process, but you want to share the table definition. DB2 does not perform logging and locking operations for created temporary tables. Therefore, SQL statements that use these tables can execute queries efficiently.

Each application process has its own instance of the created temporary table.

You create the definition of a created temporary table using the SQL CREATE GLOBAL TEMPORARY TABLE statement.

Example: The following statement creates the definition of a table called TEMPPROD:

```

CREATE GLOBAL TEMPORARY TABLE TEMPPROD
(SERIAL      CHAR(8)      NOT NULL,
DESCRIPTION  VARCHAR(60) NOT NULL,
MFGCOST      DECIMAL(8,2),
MFGDEPT      CHAR(3),
MARKUP       SMALLINT,
SALESDEPT    CHAR(3),
CURDATE      DATE        NOT NULL);

```

Example: You can also create this same definition by copying the definition of a base table (named PROD) by using the LIKE clause:

```

CREATE GLOBAL TEMPORARY TABLE TEMPPROD LIKE PROD;

```

The SQL statements in the previous examples create identical definitions for the TEMPPROD table, but these tables differ slightly from the PROD sample table PROD. The PROD sample table contains two columns, DESCRIPTION and

CURDATE, that are defined as NOT NULL WITH DEFAULT. Because created temporary tables do not support non-null default values, the DESCRIPTION and CURDATE columns in the TEMPPROD table are defined as NOT NULL and do not have defaults.

After you run one of the two CREATE statements, the definition of TEMPPROD exists, but no instances of the table exist. To create an instance of TEMPPROD, you must use TEMPPROD in an application. DB2 creates an instance of the table when TEMPPROD is specified in one of the following SQL statements:

- OPEN
- SELECT
- INSERT
- DELETE

Restriction: You cannot use the MERGE statement with created temporary tables.

An instance of a created temporary table exists at the current server until one of the following actions occurs:

- The application process ends.
- The remote server connection through which the instance was created terminates.
- The unit of work in which the instance was created completes.

When you run a ROLLBACK statement, DB2 deletes the instance of the created temporary table. When you run a COMMIT statement, DB2 deletes the instance of the created temporary table unless a cursor for accessing the created temporary table is defined with the WITH HOLD clause and is open.

Example: Suppose that you create a definition of TEMPPROD and then run an application that contains the following statements:

```
EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM TEMPPROD;
EXEC SQL INSERT INTO TEMPPROD SELECT * FROM PROD;
EXEC SQL OPEN C1;
:
:
EXEC SQL COMMIT;
:
EXEC SQL CLOSE C1;
```

When you run the INSERT statement, DB2 creates an instance of TEMPPROD and populates that instance with rows from table PROD. When the COMMIT statement runs, DB2 deletes all rows from TEMPPROD. However, assume that you change the declaration of cursor C1 to the following declaration:

```
EXEC SQL DECLARE C1 CURSOR WITH HOLD
FOR SELECT * FROM TEMPPROD;
```

In this case, DB2 does not delete the contents of TEMPPROD until the application ends because C1, a cursor that is defined with the WITH HOLD clause, is open when the COMMIT statement runs. In either case, DB2 drops the instance of TEMPPROD when the application ends.

To drop the definition of TEMPPROD, you must run the following statement:

```
DROP TABLE TEMPPROD;
```

Temporary tables

Use temporary tables when you need to store data for only the duration of an application process. Depending on whether you want to share the table definition, you can create a created temporary table or a declared temporary table.

The two kinds of temporary tables are:

- Created temporary tables, which you define using a `CREATE GLOBAL TEMPORARY TABLE` statement
- Declared temporary tables, which you define using a `DECLARE GLOBAL TEMPORARY TABLE` statement

SQL statements that use temporary tables can run faster because of the following reasons:

- DB2 does no logging (for created temporary tables) or limited logging (for declared temporary tables).
- For created temporary tables, DB2 provides no locking. For declared temporary tables, DB2 provides limited locking.

Temporary tables are especially useful when you need to sort or query intermediate result tables that contain a large number of rows, but you want to store only a small subset of those rows permanently.

Temporary tables can also return result sets from stored procedures. The following topics provide more details about created temporary tables and declared temporary tables:

- “Creating created temporary tables” on page 479
- “Creating declared temporary tables”

For more information, see “Writing an external procedure to return result sets to a distributed client” on page 641.

Creating declared temporary tables

Use declared temporary tables when you need to store data for only the life of an application process and do not need to share the table definition. The definition of this table exists only while the application process runs. DB2 performs limited logging and locking operations for declared temporary tables.

You create an instance of a declared temporary table by using the SQL `DECLARE GLOBAL TEMPORARY TABLE` statement. That instance is known only to the application process in which the table is declared, so you can declare temporary tables with the same name in different applications. The qualifier for a declared temporary table is `SESSION`.

Before you can define declared temporary tables, you must have a `WORKFILE` database that has at least one table space with a 32-KB page size.

To create a declared temporary table, specify the `DECLARE GLOBAL TEMPORARY TABLE` statement. In that statement, specify the columns that the table is to contain by performing one of the following actions:

- Specify all the columns in the table.
Unlike columns of created temporary tables, columns of declared temporary tables can include the `WITH DEFAULT` clause.
- Use a `LIKE` clause to copy the definition of a base table, created temporary table, or view.

If the base table, created temporary table, or view from which you select columns has identity columns, you can specify that the corresponding columns in the declared temporary table are also identity columns. To include these identity columns, specify the `INCLUDING IDENTITY COLUMN ATTRIBUTES` clause when you define the declared temporary table.

If the source table has a row change timestamp column, you can specify that those column attributes are inherited in the declared temporary table by specifying `INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES`.

- Use a fullselect to choose specific columns from a base table, created temporary table, or view.

If you want the declared temporary table columns to inherit the defaults for columns of the table or view that is named in the fullselect, specify the `INCLUDING COLUMN DEFAULTS` clause. If you want the declared temporary table columns to have default values that correspond to their data types, specify the `USING TYPE DEFAULTS` clause.

Example: The following statement defines a declared temporary table called `TEMPPROD` by explicitly specifying the columns.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
(SERIAL      CHAR(8)      NOT NULL WITH DEFAULT '99999999',
DESCRIPTION VARCHAR(60) NOT NULL,
PRODCOUNT   INTEGER GENERATED ALWAYS AS IDENTITY,
MFGCOST     DECIMAL(8,2),
MFGDEPT     CHAR(3),
MARKUP      SMALLINT,
SALESDEPT   CHAR(3),
CURDATE     DATE         NOT NULL);
```

Example: The following statement defines a declared temporary table called `TEMPPROD` by copying the definition of a base table. The base table has an identity column that the declared temporary table also uses as an identity column.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD LIKE BASEPROD
INCLUDING IDENTITY COLUMN ATTRIBUTES;
```

Example: The following statement defines a declared temporary table called `TEMPPROD` by selecting columns from a view. The view has an identity column that the declared temporary table also uses as an identity column. The declared temporary table inherits its default column values from the default column values of a base table on which the view is based.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
AS (SELECT * FROM PRODVIEWS)
DEFINITION ONLY
INCLUDING IDENTITY COLUMN ATTRIBUTES
INCLUDING COLUMN DEFAULTS;
```

After you run a `DECLARE GLOBAL TEMPORARY TABLE` statement, the definition of the declared temporary table exists as long as the application process runs.

If you need to delete the definition before the application process completes, you can do that with the `DROP TABLE` statement. For example, to drop the definition of `TEMPPROD`, run the following statement:

```
DROP TABLE SESSION.TEMPPROD;
```

DB2 creates an empty instance of a declared temporary table when it runs the `DECLARE GLOBAL TEMPORARY TABLE` statement. You can then perform the following actions:

- Populate the declared temporary table by using INSERT statements
- Modify the table using searched or positioned UPDATE or DELETE statements
- Query the table using SELECT statements
- Create indexes on the declared temporary table

The ON COMMIT clause that you specify in the DECLARE GLOBAL TEMPORARY TABLE statement determines whether DB2 keeps or deletes all the rows from the table when you run a COMMIT statement in an application with a declared temporary table. ON COMMIT DELETE ROWS, which is the default, causes all rows to be deleted from the table at a commit point, unless a held cursor is open on the table at the commit point. ON COMMIT PRESERVE ROWS causes the rows to remain past the commit point.

Example: Suppose that you run the following statement in an application program:

```
EXEC SQL DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
  AS (SELECT * FROM BASEPROD)
  DEFINITION ONLY
  INCLUDING IDENTITY COLUMN ATTRIBUTES
  INCLUDING COLUMN DEFAULTS
  ON COMMIT PRESERVE ROWS;
EXEC SQL INSERT INTO SESSION.TEMPPROD SELECT * FROM BASEPROD;
:
:
EXEC SQL COMMIT;
:
:
```

When DB2 runs the preceding DECLARE GLOBAL TEMPORARY TABLE statement, DB2 creates an empty instance of TEMPPROD. The INSERT statement populates that instance with rows from table BASEPROD. The qualifier, SESSION, must be specified in any statement that references TEMPPROD. When DB2 executes the COMMIT statement, DB2 keeps all rows in TEMPPROD because TEMPPROD is defined with ON COMMIT PRESERVE ROWS. When the program ends, DB2 drops TEMPPROD.

Related reference:

 [DECLARE GLOBAL TEMPORARY TABLE \(DB2 SQL\)](#)

Providing a unique key for a table

Use ROWID columns or identity columns to store unique values for each row in a table.

Question: How can I provide a unique identifier for a table that has no unique column?

Answer: Add a column with the data type ROWID or an identity column. ROWID columns and identity columns contain a unique value for each row in the table. You can define the column as GENERATED ALWAYS, which means that you cannot insert values into the column, or GENERATED BY DEFAULT, which means that DB2 generates a value if you do not specify one. If you define the ROWID or identity column as GENERATED BY DEFAULT, you need to define a unique index that includes only that column to guarantee uniqueness.

Fixing tables with incomplete definitions

If a table has an incomplete definition, you cannot load the table, insert data, retrieve data, update data, or delete data. You can however drop the table, create the primary index, and drop or create other indexes.

To check if a table has an incomplete definition, look at the STATUS column in SYSIBM.SYSTABLES. The value I indicates that the definition is incomplete.

A table definition is incomplete in any of the following circumstances:

- **If the table is defined with a primary or unique key** and all of the following conditions are true:
 - The table space for the table was explicitly created.
 - The statement is not being run with schema processor.
 - The table does not have a primary or unique index for the defined primary or unique key.
- **If the table has a ROWID column that is defined as generated by default** and all of the following conditions are true:
 - The table space for the table was explicitly created.
 - The SET CURRENT RULES special register is not set to STD.
 - No unique index is defined on the ROWID column.
- **If the table has a LOB column** and all of the following conditions are true:
 - The table space for the table was explicitly created.
 - The SET CURRENT RULES special register is not set to STD.
 - No all auxiliary LOB objects are defined for the LOB column.

You can complete the table definition by performing one of the following actions, depending on why the table definition was incomplete:

- Creating a primary index or altering the table to drop the primary key.
- Creating a unique index on the unique key or altering the table to drop the unique key.
- Defining a unique index on the ROWID column.
- Creating the necessary LOB objects.

Example of creating a primary index: To create the primary index for the project activity table, issue the following SQL statement:

```
CREATE UNIQUE INDEX XPROJAC1
  ON DSN8A10.PROJACT (PROJNO, ACTNO, ACSTDATE);
```

Dropping tables

When you drop a table, you delete the data and the table definition. You also delete all synonyms, views, indexes, referential constraints, and check constraints that are associated with that table.

The following SQL statement drops the YEMP table:

```
DROP TABLE YEMP;
```

Use the DROP TABLE statement with care: Dropping a table is **not** equivalent to deleting all its rows. When you drop a table, you lose more than its data and its

definition. You lose all synonyms, views, indexes, and referential and check constraints that are associated with that table. You also lose all authorities that are granted on the table.

Related reference:

 DROP (DB2 SQL)

Defining a view

A *view* is a named specification of a result table. Use views to control which users have access to certain data or to simplify writing SQL statements.

Use the CREATE VIEW statement to define a view and give the view a name, just as you do for a table. The view that is created with the following statement shows each department manager's name with the department data in the DSN8A10.DEPT table.

```
CREATE VIEW VDEPTM AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
     FROM DSN8A10.DEPT, DSN8A10.EMP
     WHERE DSN8A10.EMP.EMPNO = DSN8A10.DEPT.MGRNO;
```

When a program accesses the data that is defined by a view, DB2 uses the view definition to return a set of rows that the program can access with SQL statements.

Example: To see the departments that are administered by department D01 and the managers of those departments, run the following statement, which returns information from the VDEPTM view:

```
SELECT DEPTNO, LASTNAME
   FROM VDEPTM
   WHERE ADMRDEPT = 'D01';
```

When you create a view, you can reference the SESSION_USER and CURRENT SQLID special registers in the CREATE VIEW statement. When referencing the view, DB2 uses the value of the SESSION_USER or CURRENT SQLID special register that belongs to the user of the SQL statement (SELECT, UPDATE, INSERT, or DELETE) rather than the creator of the view. In other words, a reference to a special register in a view definition refers to its run time value.

A column in a view might be based on a column in a base table that is an identity column. The column in the view is also an identity column, **except** under any of the following circumstances:

- The column appears more than once in the view.
- The view is based on a join of two or more tables.
- The view is based on the union of two or more tables.
- Any column in the view is derived from an expression that refers to an identity column.


You can use views to limit access to certain kinds of data, such as salary information. Alternatively, you can use the IMPLICITLY HIDDEN clause of a CREATE TABLE statement define a column of a table to be hidden from some operations.

You can also use views for the following actions:

- Make a subset of a table's data available to an application. For example, a view based on the employee table might contain rows only for a particular department.

- Combine columns from two or more tables and make the combined data available to an application. By using a SELECT statement that matches values in one table with those in another table, you can create a view that presents data from both tables. However, you can **only select** data from this type of view. **You cannot update, delete, or insert data using a view that joins two or more tables.**
- Combine rows from two or more tables and make the combined data available to an application. By using two or more subselects that are connected by a set operator such as UNION, you can create a view that presents data from several tables. However, you can **only select** data from this type of view. **You cannot update, delete, or insert data using a view that contains UNION operations.**
- Present computed data, and make the resulting data available to an application. You can compute such data using any function or operation that you can use in a SELECT statement.

Related information:

 [Implementing DB2 views \(DB2 Administration Guide\)](#)

Views

A view does not contain data; it is a stored definition of a set of rows and columns. A view can present any or all of the data in one or more tables.

Although you cannot modify an existing view, you can drop it and create a new one if your base tables change in a way that affects the view. Dropping and creating views does not affect the base tables or their data.

Restrictions when changing data through a view

Some views are read-only and thus cannot be used to update the table data. For those views that are updatable, several restrictions apply.

Consider the following restrictions when changing data through a view:

- You must have the appropriate authorization to insert, update, or delete rows using the view.
- When you use a view to insert a row into a table, the view definition must specify all the columns in the base table that do not have a default value. The row that is being inserted must contain a value for each of those columns.
- Views that you can use to update data are subject to the same referential constraints and check constraints as the tables that you used to define the views. You can use the WITH CHECK option of the CREATE VIEW statement to specify the constraint that every row that is inserted or updated through the view must conform to the definition of the view. You can select every row that is inserted or updated through a view that is created with the WITH CHECK option.

For complex views, you can make insert, update and delete operations possible by defining INSTEAD OF triggers.

Related tasks:

[“Inserting, updating, and deleting data in views by using INSTEAD OF triggers” on page 503](#)

Related reference:

 [CREATE VIEW \(DB2 SQL\)](#)

Dropping a view

When you drop a view, you also drop all views that are defined on that view. The base table is not affected.

The following SQL statement drops the VDEPTM view:

```
DROP VIEW VDEPTM;
```

Creating a common table expression

Creating a common table expression saves you the overhead of creating and dropping a regular view that you need to use only once. Also, during statement preparation, DB2 does not need to access the catalog for the view, which saves you additional overhead.

Use the WITH clause to create a common table expression.

You can use a common table expression in a SELECT statement by using the WITH clause at the beginning of the statement.

Example: WITH clause in a SELECT statement: The following statement finds the department with the highest total pay. The query involves two levels of aggregation. First, you need to determine the total pay for each department by using the SUM function and order the results by using the GROUP BY clause. You then need to find the department with highest total pay based on the total pay for each department.

```
WITH DTOTAL (workdept, totalpay) AS
  (SELECT deptno, sum(salary+bonus)
   FROM DSN8810.EMP
   GROUP BY workdept)
SELECT workdept
   FROM DTOTAL
  WHERE totalpay = (SELECT max(totalpay)
                   FROM DTOTAL);
```

The result table for the common table expression, DTOTAL, contains the department number and total pay for each department in the employee table. The fullselect in the previous example uses the result table for DTOTAL to find the department with the highest total pay. The result table for the entire statement looks similar to the following results:

```
WORKDEPT
=====
D11
```

Using common table expressions with views:

You can use common table expressions before a fullselect in a CREATE VIEW statement. This technique is useful if you need to use the results of a common table expression in more than one query.

Example: Using a WITH clause in a CREATE VIEW statement: The following statement finds the departments that have a greater-than-average total pay and saves the results as the view RICH_DEPT:

```
CREATE VIEW RICH_DEPT (workdept) AS
  WITH DTOTAL (workdept, totalpay) AS
  (SELECT workdept, sum(salary+bonus)
   FROM DSN8A10.EMP
```

```

        GROUP BY workdept)
SELECT workdept
       FROM DTOTAL
       WHERE totalpay > (SELECT AVG(totalpay)
                        FROM DTOTAL);

```

The fullselect in the previous example uses the result table for DTOTAL to find the departments that have a greater-than-average total pay. The result table is saved as the RICH_DEPT view and looks similar to the following results:

```

WORKDEPT
=====
A00
D11
D21

```

Using common table expressions when you use INSERT:

You can use common table expressions before a fullselect in an INSERT statement.

Example: Using a common table expression in an INSERT statement: The following statement uses the result table for VITALDEPT to find the manager's number for each department that has a greater-than-average number of senior engineers. Each manager's number is then inserted into the vital_mgr table.

```

INSERT INTO vital_mgr (mgrno)
  WITH VITALDEPT (workdept, se_count) AS
  (SELECT workdept, count(*)
   FROM DSN8A10.EMP
   WHERE job = 'senior engineer'
   GROUP BY workdept)
SELECT d.manager
  FROM DSN8A10.DEPT d, VITALDEPT s
 WHERE d.workdept = s.workdept
       AND s.se_count > (SELECT AVG(se_count)
                        FROM VITALDEPT);

```

Common table expressions

A *common table expression* is like a temporary view that is defined and used for the duration of an SQL statement.

You can define a common table expression wherever you can have a fullselect statement. For example, you can include a common table expression in a SELECT, INSERT, SELECT INTO, or CREATE VIEW statement.

Each common table expression must have a unique name and be defined only once. However, you can reference a common table expression many times in the same SQL statement. Unlike regular views or nested table expressions, which derive their result tables for each reference, all references to common table expressions in a given statement share the same result table.

You can use a common table expression in the following situations:

- When you want to avoid creating a view (when general use of the view is not required, and positioned updates or deletes are not used)
- When the result table is based on host variables
- When the same result table needs to be shared in a fullselect
- When the results need to be derived using recursion

Examples of recursive common table expressions

Recursive SQL is very useful in bill of materials (BOM) applications.

Consider a table of parts with associated subparts and the quantity of subparts required by each part. For more information about recursive SQL, refer to “Creating recursive SQL by using common table expressions” on page 707.

For the examples in this topic, create the following table:

```
CREATE TABLE PARTLIST
(PART VARCHAR(8),
 SUBPART VARCHAR(8),
 QUANTITY INTEGER);
```

Assume that the PARTLIST table is populated with the values that are in the following table:

Table 80. PARTLIST table

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	14	8
07	12	8

Example 1: Single level explosion:

Single level explosion answers the question, "What parts are needed to build the part identified by '01'?". The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
(SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
 FROM PARTLIST ROOT
 WHERE ROOT.PART = '01'
 UNION ALL
 SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
 FROM RPL PARENT, PARTLIST CHILD
```

```

WHERE PARENT.SUBPART = CHILD.PART)
SELECT DISTINCT PART, SUBPART, QUANTITY
FROM RPL
ORDER BY PART, SUBPART, QUANTITY;

```

The preceding query includes a common table expression, identified by the name RPL, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the initialization fullselect, gets the direct subparts of part '01'. The FROM clause of this fullselect refers to the source table and will never refer to itself (RPL in this case). The result of this first fullselect goes into the common table expression RPL. As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses RPL to compute subparts of subparts by using the FROM clause to refer to the common table expression RPL and the source table PARTLIST with a join of a part from the source table (child) to a subpart of the current result contained in RPL (parent). The result goes then back to RPL again. The second operand of UNION is used repeatedly until no more subparts exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is shown in the following table:

Table 81. Result table for example 1

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	12	8
07	14	8

Observe in the result that part '01' contains subpart '02' which contains subpart '06' and so on. Further, notice that part '06' is reached twice, once through part '01' directly and another time through part '02'. In the output, however, the subparts of part '06' are listed only once (this is the result of using a SELECT DISTINCT).

Remember that with recursive common table expressions it is possible to introduce an infinite loop. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as follows:

```
WHERE PARENT.SUBPART = CHILD.SUBPART
```

This infinite loop is created by not coding what is intended. You should carefully determine what to code so that there is a definite end of the recursion cycle.

The result produced by this example could be produced in an application program without using a recursive common table expression. However, such an application would require coding a different query for every level of recursion. Furthermore, the application would need to put all of the results back in the database to order the final result. This approach complicates the application logic and does not perform well. The application logic becomes more difficult and inefficient for other bill of material queries, such as summarized and indented explosion queries.

Example 2: Summarized explosion:

A summarized explosion answers the question, "What is the total quantity of each part required to build part '01'?" The main difference from a single level explosion is the need to aggregate the quantities. A single level explosion indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of each subpart is needed to build part '01'.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
(
    SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
      FROM PARTLIST ROOT
     WHERE ROOT.PART = '01'
    UNION ALL
    SELECT PARENT.PART, CHILD.SUBPART,
           PARENT.QUANTITY*CHILD.QUANTITY
      FROM RPL PARENT, PARTLIST CHILD
     WHERE PARENT.SUBPART = CHILD.PART
)
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
  FROM RPL
 GROUP BY PART, SUBPART
 ORDER BY PART, SUBPART;
```

In the preceding query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name RPL, shows the aggregation of the quantity. To determine how many of each subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping the parts and subparts in the common table expression RPL and using the SUM column function in the select list of the main fullselect.

The result of the query is shown in the following table:

Table 82. Result table for example 2

PART	SUBPART	Total QTY Used
01	02	2
01	03	3
01	04	4
01	05	14

Table 82. Result table for example 2 (continued)

PART	SUBPART	Total QTY Used
01	06	15
01	07	18
01	08	40
01	09	44
01	10	140
01	11	140
01	12	294
01	13	150
01	14	144

Consider the total quantity for subpart '06'. The value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed two times by part '01'.

Example 3: Controlling depth:

You can control the depth of a recursive query to answer the question, "What are the first two levels of parts that are needed to build part '01'?" For the sake of clarity in this example, the level of each part is included in the result table.

```
WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS
(
    SELECT 1, ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
    FROM PARTLIST ROOT
    WHERE ROOT.PART = '01'
    UNION ALL
    SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
    FROM RPL PARENT, PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART
    AND PARENT.LEVEL < 2
)
SELECT PART, LEVEL, SUBPART, QUANTITY
FROM RPL;
```

This query is similar to the query in example 1. The column LEVEL is introduced to count the level each subpart is from the original part. In the initialization fullselect, the value for the LEVEL column is initialized to 1. In the subsequent fullselect, the level from the parent table increments by 1. To control the number of levels in the result, the second fullselect includes the condition that the level of the parent must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is shown in the following table:

Table 83. Result table for example 3

PART	LEVEL	SUBPART	QUANTITY
01	1	02	2
01	1	03	3
01	1	04	4
01	1	06	3
02	2	05	7

Table 83. Result table for example 3 (continued)

PART	LEVEL	SUBPART	QUANTITY
02	2	06	6
03	2	07	6
04	2	08	10
04	2	09	11
06	2	12	10
06	2	13	10

Creating triggers

A *trigger* is a set of SQL statements that execute when a certain event occurs in a table. Use triggers to control changes in DB2 databases. Triggers are more powerful than constraints because they can monitor a broader range of changes and perform a broader range of actions.

Using triggers for active data:

For example, a constraint can disallow an update to the salary column of the employee table if the new value is over a certain amount. A trigger can monitor the amount by which the salary changes, as well as the salary value. If the change is above a certain amount, the trigger might substitute a valid value and call a user-defined function to send a notice to an administrator about the invalid update.

Triggers also move application logic into DB2, which can result in faster application development and easier maintenance. For example, you can write applications to control salary changes in the employee table, but each application program that changes the salary column must include logic to check those changes. A better method is to define a trigger that controls changes to the salary column. Then DB2 does the checking for any application that modifies salaries.

Example of creating and using a trigger:

Triggers automatically execute a set of SQL statements whenever a specified event occurs. These SQL statements can perform tasks such as validation and editing of table changes, reading and modifying tables, or invoking functions or stored procedures that perform operations both inside and outside DB2.

You create triggers using the CREATE TRIGGER statement. The following figure shows an example of a CREATE TRIGGER statement.

```

1
CREATE TRIGGER REORDER
2 AFTER 3 UPDATE OF ON_HAND, MAX_STOCKED ON 4 PARTS
5
REFERENCING NEW AS N_ROW
6
7 FOR EACH ROW MODE DB2SQL
8 WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
BEGIN ATOMIC

```

```

CALL ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                        N_ROW.ON_HAND,
                        N_ROW.PARTNO);
END

```

The parts of this trigger are:

- 1** Trigger name (REORDER)
- 2** Trigger activation time (AFTER)
- 3** Triggering event (UPDATE)
- 4** Subject table name (PARTS)
- 5** New transition variable correlation name (N_ROW)
- 6** Granularity (FOR EACH ROW)
- 7** Trigger condition (WHEN...)
- 8** Trigger body (BEGIN ATOMIC...END;)

When you execute this CREATE TRIGGER statement, DB2 creates a trigger package called REORDER and associates the trigger package with table PARTS. DB2 records the timestamp when it creates the trigger. If you define other triggers on the PARTS table, DB2 uses this timestamp to determine which trigger to activate first. The trigger is now ready to use.

After DB2 updates columns ON_HAND or MAX_STOCKED in any row of table PARTS, trigger REORDER is activated. The trigger calls a stored procedure called ISSUE_SHIP_REQUEST if, after a row is updated, the quantity of parts on hand is less than 10% of the maximum quantity stocked. In the trigger condition, the qualifier N_ROW represents a value in a modified row after the triggering event.

When you no longer want to use trigger REORDER, you can delete the trigger by executing the statement:

```
DROP TRIGGER REORDER;
```

Executing this statement drops trigger REORDER and its associated trigger package named REORDER.

If you drop table PARTS, DB2 also drops trigger REORDER and its trigger package.

Parts of a trigger:

A trigger contains the following parts:

- trigger name
- subject table
- trigger activation time
- triggering event
- granularity
- transition variables
- transition tables
- triggered action

Trigger name:

Use an ordinary identifier to name your trigger. You can use a qualifier or let DB2 determine the qualifier. When DB2 creates a trigger package for the trigger, it uses the qualifier for the collection ID of the trigger package. DB2 uses these rules to determine the qualifier:

- If you use static SQL to execute the CREATE TRIGGER statement, DB2 uses the authorization ID in the bind option QUALIFIER for the package that contains the CREATE TRIGGER statement. If the bind command does not include the QUALIFIER option, DB2 uses the owner of the package.
- If you use dynamic SQL to execute the CREATE TRIGGER statement, DB2 uses the authorization ID in special register CURRENT SCHEMA.

Subject table:

When you perform an insert, update, or delete operation on this table, the trigger is activated. You must name a local table in the CREATE TRIGGER statement. You cannot define a trigger on a catalog table or on a view.

Trigger activation time:

The two choices for trigger activation time are NO CASCADE BEFORE and AFTER. NO CASCADE BEFORE means that the trigger is activated before DB2 makes any changes to the subject table, and that the triggered action does not activate any other triggers. AFTER means that the trigger is activated after DB2 makes changes to the subject table and can activate other triggers. Triggers with an activation time of NO CASCADE BEFORE are known as before triggers. Triggers with an activation time of AFTER are known as after triggers.

Triggering event:

Every trigger is associated with an event. A trigger is activated when the triggering event occurs in the subject table. The triggering event is one of the following SQL operations:

- insert
- update
- delete

A triggering event can also be an update or delete operation that occurs as the result of a referential constraint with ON DELETE SET NULL or ON DELETE CASCADE.

Triggers are not activated as the result of updates made to tables by DB2 utilities, with the exception of the LOAD utility when it is specified with the RESUME YES and SHRLEVEL CHANGE options.

When the triggering event for a trigger is an update operation, the trigger is called an update trigger. Similarly, triggers for insert operations are called insert triggers, and triggers for delete operations are called delete triggers.

The SQL statement that performs the triggering SQL operation is called the triggering SQL statement. Each triggering event is associated with one subject table and one SQL operation.

The following trigger is defined with an insert triggering event:

```

CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END

```

If the triggering SQL operation is an update operation, the event can be associated with specific columns of the subject table. In this case, the trigger is activated only if the update operation updates any of the specified columns.

The following trigger, PAYROLL1, which invokes user-defined function named PAYROLL_LOG, is activated only if an update operation is performed on the SALARY or BONUS column of table PAYROLL:

```

CREATE TRIGGER PAYROLL1
  AFTER UPDATE OF SALARY, BONUS ON PAYROLL
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES(PAYROLL_LOG(USER, 'UPDATE', CURRENT TIME, CURRENT DATE));
  END

```

Granularity:

The triggering SQL statement might modify multiple rows in the table. The granularity of the trigger determines whether the trigger is activated only once for the triggering SQL statement or once for every row that the SQL statement modifies. The granularity values are:

- FOR EACH ROW

The trigger is activated once for each row that DB2 modifies in the subject table. If the triggering SQL statement modifies no rows, the trigger is not activated. However, if the triggering SQL statement updates a value in a row to the same value, the trigger is activated. For example, if an UPDATE trigger is defined on table COMPANY_STATS, the following SQL statement will activate the trigger.

```
UPDATE COMPANY_STATS SET NBEMP = NBEMP;
```

- FOR EACH STATEMENT

The trigger is activated once when the triggering SQL statement executes. The trigger is activated even if the triggering SQL statement modifies no rows.

Triggers with a granularity of FOR EACH ROW are known as row triggers. Triggers with a granularity of FOR EACH STATEMENT are known as statement triggers. Statement triggers can only be after triggers.

The following statement is an example of a row trigger:

```

CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END

```

Trigger NEW_HIRE is activated once for every row inserted into the employee table.

Transition variables:

When you code a row trigger, you might need to refer to the values of columns in each updated row of the subject table. To do this, specify transition variables in the REFERENCING clause of your CREATE TRIGGER statement. The two types of transition variables are:

- Old transition variables, specified with the OLD *transition-variable* clause, capture the values of columns before the triggering SQL statement updates them. You can define old transition variables for update and delete triggers.
- New transition variables, specified with the NEW *transition-variable* clause, capture the values of columns after the triggering SQL statement updates them. You can define new transition variables for update and insert triggers.

The following example uses transition variables and invocations of the IDENTITY_VAL_LOCAL function to access values that are assigned to identity columns.

Suppose that you have created tables T and S, with the following definitions:

```
CREATE TABLE T
  (ID SMALLINT GENERATED BY DEFAULT AS IDENTITY (START WITH 100),
   C2 SMALLINT,
   C3 SMALLINT,
   C4 SMALLINT);
CREATE TABLE S
  (ID SMALLINT GENERATED ALWAYS AS IDENTITY,
   C1 SMALLINT);
```

Define a before insert trigger on T that uses the IDENTITY_VAL_LOCAL built-in function to retrieve the current value of identity column ID, and uses transition variables to update the other columns of T with the identity column value.

```
CREATE TRIGGER TR1
  NO CASCADE BEFORE INSERT
  ON T REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET N.C3 =N.ID;
    SET N.C4 =IDENTITY_VAL_LOCAL();
    SET N.ID =N.C2 *10;
    SET N.C2 =IDENTITY_VAL_LOCAL();
  END
```

Now suppose that you execute the following INSERT statement:

```
INSERT INTO S (C1) VALUES (5);
```

This statement inserts a row into S with a value of 5 for column C1 and a value of 1 for identity column ID. Next, suppose that you execute the following SQL statement, which activates trigger TR1:

```
INSERT INTO T (C2)
  VALUES (IDENTITY_VAL_LOCAL());
```

This insert statement, and the subsequent activation of trigger TR1, have the following results:

- The INSERT statement obtains the most recent value that was assigned to an identity column (1), and inserts that value into column C2 of table T. 1 is the value that DB2 inserted into identity column ID of table S.
- When the INSERT statement executes, DB2 inserts the value 100 into identity column ID column of C2.

- The first statement in the body of trigger TR1 inserts the value of transition variable N.ID (100) into column C3. N.ID is the value that identity column ID contains *after* the INSERT statement executes.
- The second statement in the body of trigger TR1 inserts the null value into column C4. By definition, the result of the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger is the null value.
- The third statement in the body of trigger TR1 inserts 10 times the value of transition variable N.C2 (10*1) into identity column ID of table T. N.C2 is the value that column C2 contains *after* the INSERT is executed.
- The fourth statement in the body of trigger TR1 inserts the null value into column C2. By definition, the result of the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger is the null value.

Transition tables:

If you want to refer to the entire set of rows that a triggering SQL statement modifies, rather than to individual rows, use a transition table. Like transition variables, transition tables can appear in the REFERENCING clause of a CREATE TRIGGER statement. Transition tables are valid for both row triggers and statement triggers. The two types of transition tables are:

- Old transition tables, specified with the OLD TABLE *transition-table-name* clause, capture the values of columns before the triggering SQL statement updates them. You can define old transition tables for update and delete triggers.
- New transition tables, specified with the NEW TABLE *transition-table-name* clause, capture the values of columns after the triggering SQL statement updates them. You can define new transition variables for update and insert triggers.

The scope of old and new transition table names is the trigger body. If another table exists that has the same name as a transition table, any unqualified reference to that name in the trigger body points to the transition table. To reference the other table in the trigger body, you must use the fully qualified table name.

The following example uses a new transition table to capture the set of rows that are inserted into the INVOICE table:

```
CREATE TRIGGER LRG_ORDR
  AFTER INSERT ON INVOICE
  REFERENCING NEW TABLE AS N_TABLE
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    SELECT LARGE_ORDER_ALERT(CUST_NO,
      TOTAL_PRICE, DELIVERY_DATE)
    FROM N_TABLE WHERE TOTAL_PRICE > 10000;
  END
```

The SELECT statement in LRG_ORDER causes user-defined function LARGE_ORDER_ALERT to execute for each row in transition table N_TABLE that satisfies the WHERE clause (TOTAL_PRICE > 10000).

Triggered action:

When a trigger is activated, a triggered action occurs. Every trigger has one triggered action, which consists of a trigger condition and a trigger body.

Trigger condition:

If you want the triggered action to occur only when certain conditions are true, code a trigger condition. A trigger condition is similar to a predicate in a SELECT, except that the trigger condition begins with WHEN, rather than WHERE. If you do not include a trigger condition in your triggered action, the trigger body executes every time the trigger is activated.

For a row trigger, DB2 evaluates the trigger condition once for each modified row of the subject table. For a statement trigger, DB2 evaluates the trigger condition once for each execution of the triggering SQL statement.

If the trigger condition of a before trigger has a fullselect, the fullselect cannot reference the subject table.

The following example shows a trigger condition that causes the trigger body to execute only when the number of ordered items is greater than the number of available items:

```
CREATE TRIGGER CK_AVAIL
NO CASCADE BEFORE INSERT ON ORDERS
REFERENCING NEW AS NEW_ORDER
FOR EACH ROW MODE DB2SQL
WHEN (NEW_ORDER.QUANTITY >
      (SELECT ON_HAND FROM PARTS
       WHERE NEW_ORDER.PARTNO=PARTS.PARTNO))
BEGIN ATOMIC
  VALUES (ORDER_ERROR(NEW_ORDER.PARTNO,
                      NEW_ORDER.QUANTITY));
END
```

Trigger body:

In the trigger body, you code the SQL statements that you want to execute whenever the trigger condition is true. If the trigger body consists of more than one statement, it must begin with BEGIN ATOMIC and end with END. You cannot include host variables or parameter markers in your trigger body. If the trigger body contains a WHERE clause that references transition variables, the comparison operator cannot be LIKE.

The statements you can use in a trigger body depend on the activation time of the trigger. For a list of valid SQL statements for triggers, see the "Allowable SQL statements" table in the CREATE TRIGGER (DB2 SQL) topic.

The following list provides more detailed information about SQL statements that are valid in triggers:

- fullselect, CALL, and VALUES

Use a fullselect or the VALUES statement in a trigger body to conditionally or unconditionally invoke a user-defined function. Use the CALL statement to invoke a stored procedure. See "Invoking a stored procedure or user-defined function from a trigger" on page 501 for more information on invoking user-defined functions and stored procedures from triggers.

A fullselect in the trigger body of a before trigger cannot reference the subject table.

- SIGNAL

Use the SIGNAL statement in the trigger body to report an error condition and back out any changes that are made by the trigger, as well as actions that result from referential constraints on the subject table. When DB2 executes the SIGNAL

statement, it returns an SQLCA to the application with SQLCODE -438. The SQLCA also includes the following values, which you supply in the SIGNAL statement:

- A 5-character value that DB2 uses as the SQLSTATE
- An error message that DB2 places in the SQLERRMC field

In the following example, the SIGNAL statement causes DB2 to return an SQLCA with SQLSTATE 75001 and terminate the salary update operation if an employee's salary increase is over 20%:

```
CREATE TRIGGER SAL_ADJ
  BEFORE UPDATE OF SALARY ON EMP
  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
```

- *SET transition-variable*

Because before triggers operate on rows of a table before those rows are modified, you cannot perform operations in the body of a before trigger that directly modify the subject table. You can, however, use the SET *transition-variable* statement to modify the values in a row before those values go into the table. For example, this trigger uses a new transition variable to enter today's date for the new employee's hire date:

```
CREATE TRIGGER HIREDATE
  NO CASCADE BEFORE INSERT ON EMP
  REFERENCING NEW AS NEW_VAR
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    SET NEW_VAR.HIRE_DATE = CURRENT_DATE;
  END
```

- INSERT, DELETE (searched), UPDATE (searched), and MERGE

Because you can include INSERT, DELETE (searched), UPDATE (searched), and MERGE statements in your trigger body, execution of the trigger body might cause activation of other triggers. See "Trigger cascading" on page 505 for more information.

If any SQL statement in the trigger body fails during trigger execution, DB2 rolls back all changes that are made by the triggering SQL statement and the triggered SQL statements. However, if the trigger body executes actions that are outside of DB2's control or are not under the same commit coordination as the DB2 subsystem in which the trigger executes, DB2 cannot undo those actions. Examples of external actions that are not under DB2's control are:

- Performing updates that are not under RRS commit control
- Sending an electronic mail message

If the trigger executes external actions that are under the same commit coordination as the DB2 subsystem under which the trigger executes, and an error occurs during trigger execution, DB2 places the application process that issued the triggering statement in a must-rollback state. The application must then execute a rollback operation to roll back those external actions. Examples of external actions that are under the same commit coordination as the triggering SQL operation are:

- Executing a distributed update operation
- From a user-defined function or stored procedure, executing an external action that affects an external resource manager that is under RRS commit control.

Related reference:

 [CREATE TRIGGER \(DB2 SQL\)](#)

 [LOAD \(DB2 Utilities\)](#)

Invoking a stored procedure or user-defined function from a trigger

A trigger body can include only SQL statements. To perform actions or use logic that is not available in SQL statements, create user-defined functions or stored procedures. Then invoke them from within the trigger body.

Introductory concepts:

Triggers (Introduction to DB2 for z/OS)

Restriction: You cannot include INSERT, UPDATE, DELETE, or MERGE statements in stored procedures or user-defined functions that are invoked by a BEFORE TRIGGER. These actions are not allowed, because BEFORE triggers must not modify any table.

To invoke a stored procedure or user-defined function from a trigger:

1. Ensure that the stored procedure or user-defined function is defined before the trigger is defined.
 - Define procedures by using the CREATE PROCEDURE statement.
 - Define triggers by using the CREATE FUNCTION statement.
2. Invoke the user-defined function or stored procedure by performing one of the following actions:
 - To invoke a user-defined function, include the user-defined function in one of the following statements in the trigger:

SELECT statement

Use a SELECT statement to execute the function conditionally. The number of times that the user-defined function executes depends on the number of rows in the result table of the SELECT statement. For example, in the following trigger, the SELECT statement invokes user-defined function LARGE_ORDER_ALERT. This function executes once for each row in transition table N_TABLE with an order price of more than 10000:

```
CREATE TRIGGER LRG_ORDR
  AFTER INSERT ON INVOICE
  REFERENCING NEW TABLE AS N_TABLE
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    SELECT LARGE_ORDER_ALERT(CUST_NO, TOTAL_PRICE, DELIVERY_DATE)
      FROM N_TABLE WHERE TOTAL_PRICE > 10000;
  END
```

VALUES statement

Use the VALUES statement to execute a function unconditionally. The function executes once for each execution of a statement trigger or once for each row in a row trigger. In the following example, user-defined function PAYROLL_LOG executes every time the trigger PAYROLL1 is activated. This trigger is activated when an update operation occurs.


```

CREATE TRIGGER PAYROLL1
  AFTER UPDATE ON PAYROLL
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES(PAYROLL_LOG(USER, 'UPDATE',
      CURRENT TIME, CURRENT DATE));
  END

```

]

- To invoke a stored procedure, include a CALL statement in the trigger. The parameters of this stored procedure call must be constants, transition variables, table locators, or expressions.

If the parameter is a transition variable or table locator, and the CALL statement is in a BEFORE or AFTER trigger, DB2 returns a warning.

3. To pass transition tables from the trigger to the user-defined function or stored procedure, use table locators.

When you call a user-defined function or stored procedure from a trigger, you might want to give the function or procedure access to the entire set of modified rows. In this case, use table locators to pass a pointer to the old or new transition table.

Most of the code for using a table locator is in the function or stored procedure that receives the locator.

To pass the transition table from a trigger, specify the parameter *TABLE transition-table-name* when you invoke the function or stored procedure. This parameter causes DB2 to pass a table locator for the transition table to the user-defined function or stored procedure. For example, the following trigger passes a table locator for a transition table NEWEMPS to stored procedure CHECKEMP:

```

CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
  REFERENCING NEW TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    CALL CHECKEMP(TABLE NEWEMPS);
  END

```

Related concepts:

“User-defined functions” on page 520






Related tasks:

“Accessing transition tables in a user-defined function or stored procedure” on page 551

“Creating a stored procedure” on page 558

“Defining a user-defined function” on page 517

Related reference:

-  CALL (DB2 SQL)
-  CREATE FUNCTION (DB2 SQL)
-  CREATE PROCEDURE (DB2 SQL)
-  select-statement (DB2 SQL)
-  VALUES (DB2 SQL)

Inserting, updating, and deleting data in views by using INSTEAD OF triggers

INSTEAD OF triggers are triggers that execute instead of the INSERT, UPDATE, or DELETE statement that activates the trigger. You can define these triggers on views only. Use INSTEAD OF triggers to insert, update, and delete data in complex views.

Complex views are those views that are defined on expressions or multiple tables. In some cases, those views are read only. In these cases, INSTEAD OF triggers make the insert, update and delete operations possible. If the complex view is not read only, you can request an insert, update, or delete operation. However, DB2 automatically decides how to perform that operation on the base tables that are referenced in the view. With INSTEAD OF triggers, you can define exactly how DB2 is to execute an insert, update, or delete operation on the view. You no longer leave the decision to DB2.

To insert, update, or delete data in a view by using INSTEAD OF triggers:

1. Define one or more INSTEAD OF triggers on the view by using a CREATE TRIGGER statement.

You can create one trigger for each of the following operations: INSERT, UPDATE, and DELETE. These triggers define the action that DB2 is to take for each of these operations.

2. Submit a INSERT, UPDATE, or DELETE statement on the view.

DB2 executes the appropriate INSTEAD OF trigger.

Example: Suppose that you create the following view on the sample tables DSN8A10.EMP and DSN8A10.DEPT:

```
CREATE VIEW EMPV (EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO, HIREDATE,DEPTNAME)
  AS SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO, HIREDATE, DEPTNAME
  FROM DSN8A10.EMP, DSN8A10.DEPT WHERE DSN8A10.EMP.WORKDEPT
  = DSN8A10.DEPT.DEPTNO
```

Suppose that you also define the following three INSTEAD OF triggers:

```
CREATE TRIGGER EMPV_INSERT INSTEAD OF INSERT ON EMPV
REFERENCING NEW AS NEWEMP
FOR EACH ROW MODE DB2SQL
  INSERT INTO DSN8A10.EMP (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT,
  PHONENO, HIREDATE)
  VALUES(NEWEMP.EMPNO, NEWEMP.FIRSTNME, NEWEMP.MIDINIT, NEWEMP.LASTNAME,
  COALESCE((SELECT D.DEPTNO FROM DSN8A10.DEPT AS D
  WHERE D.DEPTNAME = NEWEMP.DEPTNAME),
  RAISE_ERROR('70001', 'Unknown department name')),
  NEWEMP.PHONENO, NEWEMP.HIREDATE)
```

```
CREATE TRIGGER EMPV_UPDATE INSTEAD OF UPDATE ON EMPV
REFERENCING NEW AS NEWEMP OLD AS OLDEMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE DSN8A10.EMP AS E
  SET (E.FIRSTNME, E.MIDINIT, E.LASTNAME, E.WORKDEPT, E.PHONENO,
  E.HIREDATE)
  = (NEWEMP.FIRSTNME, NEWEMP.MIDINIT, NEWEMP.LASTNAME,
  COALESCE((SELECT D.DEPTNO FROM DSN8A10.DEPT AS D
  WHERE D.DEPTNAME = OLDEMP.DEPTNAME),
  RAISE_ERROR ('70001', 'Unknown department name'))
  NEWEMP.PHONENO, NEWEMP.HIREDATE)
  WHERE NEWEMP.EMPNO = E.EMPNO;
  UPDATE DSN8A10.DEPT D SET D.DEPTNAME=NEWEMP.DEPTNAME
```

```

WHERE D.DEPTNAME=OLDEMP.DEPTNAME;
END

CREATE TRIGGER EMPV_DELETE INSTEAD OF DELETE ON EMPV
REFERENCING OLD AS OLDEMP
FOR EACH ROW MODE DB2SQL
DELETE FROM DSN8A10.EMP AS E WHERE E.EMPNO = OLDEMP.EMPNO

```

Because the view is on a query with an inner join, the view is read only. However, the INSTEAD OF triggers makes insert, update, and delete operations possible.

The following table describes what happens for various insert, update, and delete operations on the EMPV view.

Table 84. Results of INSTEAD OF triggers

SQL statement	Result
INSERT INTO EMPV VALUES (...)	The EMPV_INSERT trigger is activated. This trigger inserts the row into the base table DSN8A10.EMP if the department name matches a value in the WORKDEPT column in the DSN8A10.DEPT table. Otherwise, an error is returned. If a query had been used instead of a VALUES clause on the INSERT statement, the trigger body would be processed for each row from the query.
UPDATE EMPV SET DEPTNAME='PLANNING & STRATEGY' WHERE DEPTNAME='PLANNING'	The EMPV_UPDATE trigger is activated. This trigger updates the DEPTNAME column in the DSN8A10.DEPT for the any qualifying rows.
DELETE FROM EMPV WHERE HIREDATE<'1910-01-01'	The EMPV_DELETE trigger is activated. This trigger deletes the qualifying rows from the DSN8A10.EMP table.

Related reference:

 [CREATE TRIGGER \(DB2 SQL\)](#)

Trigger packages

A trigger package is a special type of package that is created only when you execute a CREATE TRIGGER statement. A trigger package executes only when its associated trigger is activated.

As with any other package, DB2 marks a trigger package invalid when you drop a table, index, or view on which the trigger package depends. DB2 executes an automatic rebind the next time the trigger is activated. However, if the automatic rebind fails, DB2 does not mark the trigger package as inoperative.

Unlike other packages, a trigger package is freed if you drop the table on which the trigger is defined, so you can re-create the trigger package only by recreating the table and the trigger.

You can use the subcommand REBIND TRIGGER PACKAGE to rebind a trigger package that DB2 has marked as inoperative. You can also use REBIND TRIGGER PACKAGE to change the option values with which DB2 originally bound the trigger package. You can change only a limited subset of the default bind options that DB2 used when creating the package.

Related reference:

 REBIND TRIGGER PACKAGE (DSN) (DB2 Commands)

Trigger cascading

When a trigger performs an SQL operation, it might modify the subject table or other tables with triggers, therefore DB2 also activates those triggers. This situation is called trigger cascading.

A trigger that is activated as the result of another trigger can be activated at the same level as the original trigger or at a different level. Two triggers, A and B, are activated at different levels if trigger B is activated after trigger A is activated and completes before trigger A completes. If trigger B is activated after trigger A is activated and completes after trigger A completes, then the triggers are at the same level.

For example, in these cases, trigger A and trigger B are activated at the same level:

- Table X has two triggers that are defined on it, A and B. A is a before trigger and B is an after trigger. An update to table X causes both trigger A and trigger B to activate.
- Trigger A updates table X, which has a referential constraint with table Y, which has trigger B defined on it. The referential constraint causes table Y to be updated, which activates trigger B.

In these cases, trigger A and trigger B are activated at different levels:

- Trigger A is defined on table X, and trigger B is defined on table Y. Trigger B is an update trigger. An update to table X activates trigger A, which contains an UPDATE statement on table B in its trigger body. This UPDATE statement activates trigger B.
- Trigger A calls a stored procedure. The stored procedure contains an INSERT statement for table X, which has insert trigger B defined on it. When the INSERT statement on table X executes, trigger B is activated.

When triggers are activated at different levels, it is called *trigger cascading*. Trigger cascading can occur only for after triggers because DB2 does not support cascading of before triggers.

To prevent the possibility of endless trigger cascading, DB2 supports only 16 levels of cascading of triggers, stored procedures, and user-defined functions. If a trigger, user-defined function, or stored procedure at the 17th level is activated, DB2 returns SQLCODE -724 and backs out all SQL changes in the 16 levels of cascading. However, as with any other SQL error that occurs during trigger execution, if any action occurs that is outside the control of DB2, that action is not backed out.

You can write a monitor program that issues IFI READS requests to collect DB2 trace information about the levels of cascading of triggers, user-defined functions, and stored procedures in your programs.

Related tasks:

 Invoking IFI from a monitor program (DB2 Performance)

Order of multiple triggers

You can create multiple triggers for the same subject table, event, and activation time. The order in which those triggers are activated is the order in which the triggers were created.

DB2 records the timestamp when each CREATE TRIGGER statement executes. When an event occurs in a table that activates more than one trigger, DB2 uses the stored timestamps to determine which trigger to activate first.

DB2 always activates all before triggers that are defined on a table before the after triggers that are defined on that table, but within the set of before triggers, the activation order is by timestamp, and within the set of after triggers, the activation order is by timestamp.

In this example, triggers NEWHIRE1 and NEWHIRE2 have the same triggering event (INSERT), the same subject table (EMP), and the same activation time (AFTER). Suppose that the CREATE TRIGGER statement for NEWHIRE1 is run before the CREATE TRIGGER statement for NEWHIRE2:

```
CREATE TRIGGER NEWHIRE1
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END

CREATE TRIGGER NEWHIRE2
  AFTER INSERT ON EMP
  REFERENCING NEW AS N_EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE DEPTS SET NBEMP = NBEMP + 1
      WHERE DEPT_ID = N_EMP.DEPT_ID;
  END
```

When an insert operation occurs on table EMP, DB2 activates NEWHIRE1 first because NEWHIRE1 was created first. Now suppose that someone drops and re-creates NEWHIRE1. NEWHIRE1 now has a later timestamp than NEWHIRE2, so the next time an insert operation occurs on EMP, NEWHIRE2 is activated before NEWHIRE1.

If two row triggers are defined for the same action, the trigger that was created earlier is activated first for all affected rows. Then the second trigger is activated for all affected rows. In the previous example, suppose that an INSERT statement with a fullselect inserts 10 rows into table EMP. NEWHIRE1 is activated for all 10 rows, then NEWHIRE2 is activated for all 10 rows.

Interactions between triggers and referential constraints

When you create triggers, you need to understand the interactions among the triggers and constraints on your tables. You also need to understand the effect that the order of processing of those constraints and triggers can have on the results.

In general, the following steps occur when triggering SQL statement S1 performs an insert, update, or delete operation on table T1:

1. DB2 determines the rows of T1 to modify. Call that set of rows M1. The contents of M1 depend on the SQL operation:
 - For a delete operation, all rows that satisfy the search condition of the statement for a searched delete operation, or the current row for a positioned delete operation
 - For an insert operation, the row identified by the VALUES statement, or the rows identified by the result table of a SELECT clause within the INSERT statement

- For an update operation, all rows that satisfy the search condition of the statement for a searched update operation, or the current row for a positioned update operation
2. DB2 processes all before triggers that are defined on T1, in order of creation. Each before trigger executes the triggered action once for each row in M1. If M1 is empty, the triggered action does not execute.
If an error occurs when the triggered action executes, DB2 rolls back all changes that are made by S1.
 3. DB2 makes the changes that are specified in statement S1 to table T1, unless an INSTEAD OF trigger is defined for that action. If an appropriate INSTEAD OF trigger is defined, DB2 executes the trigger instead of the statement and skips the remaining steps in this list.
If an error occurs, DB2 rolls back all changes that are made by S1.
 4. If M1 is not empty, DB2 applies all the following constraints and checks that are defined on table T1:
 - Referential constraints
 - Check constraints
 - Checks that are due to updates of the table through views defined WITH CHECK OPTION
 Application of referential constraints with rules of DELETE CASCADE or DELETE SET NULL are activated before delete triggers or before update triggers on the dependent tables.
If any constraint is violated, DB2 rolls back all changes that are made by constraint actions or by statement S1.
 5. DB2 processes all after triggers that are defined on T1, and all after triggers on tables that are modified as the result of referential constraint actions, in order of creation.
Each after row trigger executes the triggered action once for each row in M1. If M1 is empty, the triggered action does not execute.
Each after statement trigger executes the triggered action once for each execution of S1, even if M1 is empty.

If any triggered actions contain SQL insert, update, or delete operations, DB2 repeats steps 1 through 5 for each operation.

If an error occurs when the triggered action executes, or if a triggered action is at the 17th level of trigger cascading, DB2 rolls back all changes that are made in step 5 and all previous steps.

For example, table DEPT is a parent table of EMP, with these conditions:

- The DEPTNO column of DEPT is the primary key.
- The WORKDEPT column of EMP is the foreign key.
- The constraint is ON DELETE SET NULL.

Suppose the following trigger is defined on EMP:

```
CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
  REFERENCING NEW TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES(CHECKEMP(TABLE NEWEMPS));
  END
```

Also suppose that an SQL statement deletes the row with department number E21 from DEPT. Because of the constraint, DB2 finds the rows in EMP with a

WORKDEPT value of E21 and sets WORKDEPT in those rows to null. This is equivalent to an update operation on EMP, which has update trigger EMPRAISE. Therefore, because EMPRAISE is an after trigger, EMPRAISE is activated after the constraint action sets WORKDEPT values to null.

Interactions between triggers and tables that have multilevel security with row-level granularity

A BEFORE trigger affects the value of the transition variable that is associated with a security label column.

If a subject table has a security label column, the column in the transition table or transition variable that corresponds to the security label column in the subject table does not inherit the security label attribute. This means that the multilevel security check with row-level granularity is not enforced for the transition table or the transition variable. If you add a security label column to a subject table using the ALTER TABLE statement, the rules are the same as when you add any column to a subject table because the column in the transition table or the transition variable that corresponds to the security label column does not inherit the security label attribute.

If the ID you are using does not have write-down privilege and you execute an insert or update operation, the security label value of your ID is assigned to the security label column for the rows that you are inserting or updating.

When a BEFORE trigger is activated, the value of the transition variable that corresponds to the security label column is the security label of the ID if either of the following conditions is true:

- The user does not have write-down privilege
- The value for the security label column is not specified

If the user does not have write-down privilege, and the trigger changes the transition variable that corresponds to the security label column, the value of the security label column is changed back to the security label value of the user before the row is written to the page.

Related concepts:

 [Multilevel security \(Managing Security\)](#)

Triggers that return inconsistent results

When you create triggers and write SQL statements that activate those triggers, you need to ensure that executing those statements always produces the same results.

Two common reasons that you can get inconsistent results are:

- Positioned UPDATE or DELETE statements that use uncorrelated subqueries cause triggers to operate on a larger result table than you intended.
- DB2 does not always process rows in the same order, so triggers that propagate rows of a table can generate different result tables at different times.

The following examples demonstrate these situations.

Example: Effect of an uncorrelated subquery on a triggered action: Suppose that tables T1 and T2 look like this:

Table T1	Table T2
A1	B1
==	==
1	1
2	2

The following trigger is defined on T1:

```
CREATE TRIGGER TR1
  AFTER UPDATE OF T1
  FOR EACH ROW
  MODE DB2SQL
  BEGIN ATOMIC
    DELETE FROM T2 WHERE B1 = 2;
  END
```

Now suppose that an application executes the following statements to perform a positioned update operation:

```
EXEC SQL BEGIN DECLARE SECTION;
  long hv1;
EXEC SQL END DECLARE SECTION;
:
:
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT A1 FROM T1
  WHERE A1 IN (SELECT B1 FROM T2)
  FOR UPDATE OF A1;
:
:
EXEC SQL OPEN C1;
:
:
while(SQLCODE>=0 && SQLCODE!=100)
{
  EXEC SQL FETCH C1 INTO :hv1;
  UPDATE T1 SET A1=5 WHERE CURRENT OF C1;
}
```

When DB2 executes the FETCH statement that positions cursor C1 for the first time, DB2 evaluates the subselect, SELECT B1 FROM T2, to produce a result table that contains the two rows of column T2:

```
1
2
```

When DB2 executes the positioned UPDATE statement for the first time, trigger TR1 is activated. When the body of trigger TR1 executes, the row with value 2 is deleted from T2. However, because SELECT B1 FROM T2 is evaluated only once, when the FETCH statement is executed again, DB2 finds the second row of T1, even though the second row of T2 was deleted. The FETCH statement positions the cursor to the second row of T1, and the second row of T1 is updated. The update operation causes the trigger to be activated again, which causes DB2 to attempt to delete the second row of T2, even though that row was already deleted.

To avoid processing of the second row after it should have been deleted, use a correlated subquery in the cursor declaration:

```
DCL C1 CURSOR FOR
  SELECT A1 FROM T1 X
  WHERE EXISTS (SELECT B1 FROM T2 WHERE X.A1 = B1)
  FOR UPDATE OF A1;
```

In this case, the subquery, SELECT B1 FROM T2 WHERE X.A1 = B1, is evaluated for each FETCH statement. The first time that the FETCH statement executes, it positions the cursor to the first row of T1. The positioned UPDATE operation

activates the trigger, which deletes the second row of T2. Therefore, when the FETCH statement executes again, no row is selected, so no update operation or triggered action occurs.

Example: Effect of row processing order on a triggered action: The following example shows how the order of processing rows can change the outcome of an after row trigger.

Suppose that tables T1, T2, and T3 look like this:

Table T1	Table T2	Table T3
A1	B1	C1
==	==	==
1	(empty)	(empty)
2		

The following trigger is defined on T1:

```
CREATE TRIGGER TR1
  AFTER UPDATE ON T1
  REFERENCING NEW AS N
  FOR EACH ROW
  MODE DB2SQL
  BEGIN ATOMIC
    INSERT INTO T2 VALUES(N.C1);
    INSERT INTO T3 (SELECT B1 FROM T2);
  END
```

Now suppose that a program executes the following UPDATE statement:

```
UPDATE T1 SET A1 = A1 + 1;
```

The contents of tables T2 and T3 after the UPDATE statement executes depend on the order in which DB2 updates the rows of T1.

If DB2 updates the first row of T1 first, after the UPDATE statement and the trigger execute for the first time, the values in the three tables are:

Table T1	Table T2	Table T3
A1	B1	C1
==	==	==
2	2	2
2		

After the second row of T1 is updated, the values in the three tables are:

Table T1	Table T2	Table T3
A1	B1	C1
==	==	==
2	2	2
3	3	2
		3

However, if DB2 updates the second row of T1 first, after the UPDATE statement and the trigger execute for the first time, the values in the three tables are:

Table T1	Table T2	Table T3
A1	B1	C1
==	==	==
1	3	3
3		

After the first row of T1 is updated, the values in the three tables are:

Table T1	Table T2	Table T3
A1	B1	C1
==	==	==
2	3	3
3	2	3
		2

Sequence objects

A *sequence* is a user-defined object that generates a sequence of numeric values according to the specification with which the sequence was created. Sequences, unlike identity columns, are not associated with tables. Applications refer to a sequence object to get its current or next value.

The sequence of numeric values is generated in a monotonically ascending or descending order. The relationship between sequences and tables is controlled by the application, not by DB2.

Your application can reference a sequence object and coordinate the value as keys across multiple rows and tables. However, a table column that gets its values from a sequence object does not necessarily have unique values in that column. Even if the sequence object has been defined with the NO CYCLE clause, some other application might insert values into that table column other than values you obtain by referencing that sequence object.

DB2 always generates sequence numbers in order of request. However, in a data sharing group where the sequence values are cached by multiple DB2 members simultaneously, the sequence value assignments might not be in numeric order. Additionally, you might have gaps in sequence number values for the following reasons:

- If DB2 terminates abnormally before it assigns all the cached values
- If your application rolls back a transaction that increments the sequence
- If the statement containing NEXT VALUE fails after it increments the sequence

You create a sequence object with the CREATE SEQUENCE statement, alter it with the ALTER SEQUENCE statement, and drop it with the DROP SEQUENCE statement. You grant access to a sequence with the GRANT (privilege) ON SEQUENCE statement, and revoke access to the sequence with the REVOKE (privilege) ON SEQUENCE statement.

The values that DB2 generates for a sequence depend on how the sequence is created. The START WITH option determines the first value that DB2 generates. The values advance by the INCREMENT BY value in ascending or descending order.

The MINVALUE and MAXVALUE options determine the minimum and maximum values that DB2 generates. The CYCLE or NO CYCLE option determines whether DB2 wraps values when it has generated all values between the START WITH value and MAXVALUE if the values are ascending, or between the START WITH value and MINVALUE if the values are descending.

Keys across multiple tables: You can use the same sequence number as a key value in two separate tables by first generating the sequence value with a NEXT VALUE expression to insert the first row in the first table. You can then reference this same sequence value with a PREVIOUS VALUE expression to insert the other rows in the second table.

Example: Suppose that an ORDERS table and an ORDER_ITEMS table are defined in the following way:

```
CREATE TABLE ORDERS
  (ORDERNO    INTEGER NOT NULL,
   ORDER_DATE DATE DEFAULT,
   CUSTNO     SMALLINT
   PRIMARY KEY (ORDERNO));

CREATE TABLE ORDER_ITEMS
  (ORDERNO    INTEGER NOT NULL,
   PARTNO     INTEGER NOT NULL,
   QUANTITY   SMALLINT NOT NULL,
   PRIMARY KEY (ORDERNO,PARTNO),
   CONSTRAINT REF_ORDERNO FOREIGN KEY (ORDERNO)
     REFERENCES ORDERS (ORDERNO) ON DELETE CASCADE);
```

You create a sequence named ORDER_SEQ to use as key values for both the ORDERS and ORDER_ITEMS tables:

```
CREATE SEQUENCE ORDER_SEQ AS INTEGER
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 20;
```

You can then use the same sequence number as a primary key value for the ORDERS table and as part of the primary key value for the ORDER_ITEMS table:

```
INSERT INTO ORDERS (ORDERNO, CUSTNO)
  VALUES (NEXT VALUE FOR ORDER_SEQ, 12345);

INSERT INTO ORDER_ITEMS (ORDERNO, PARTNO, QUANTITY)
  VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 2);
```

The NEXT VALUE expression in the first INSERT statement generates a sequence number value for the sequence object ORDER_SEQ. The PREVIOUS VALUE expression in the second INSERT statement retrieves that same value because it was the sequence number most recently generated for that sequence object within the current application process.

DB2 object relational extensions

With the object extensions of DB2, you can incorporate object-oriented concepts and methodologies into your relational database by extending DB2 with richer sets of data types and functions.

With those extensions, you can store instances of object-oriented data types in columns of tables and operate on them using functions in SQL statements. In addition, you can control the types of operations that users can perform on those data types.

The object extensions that DB2 provides are:

- Large objects (LOBs)

The VARCHAR, VARGRAPHIC, and VARBINARY data types have a storage limit of 32 KB. Although this might be sufficient for small- to medium-size text data, applications often need to store large text documents. They might also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images. DB2 provides three data types to store these data objects as strings of up to 2 GB - 1 in size. The three data

types are binary large objects (BLOBs), character large objects (CLOBs), and double-byte character large objects (DBCLOBs).

For a detailed discussion of LOBs, see “Large objects (LOBs)” on page 465.

- Distinct types

A distinct type is a user-defined data type that shares its internal representation with a built-in data type but is considered to be a separate and incompatible type for semantic purposes. For example, you might want to define a picture type or an audio type, both of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

For a detailed discussion of distinct types, see “Distinct types.”

- User-defined functions

The built-in functions that are supplied with DB2 are a useful set of functions, but they might not satisfy all of your requirements. For those cases, you can use user-defined functions. For example, a built-in function might perform a calculation you need, but the function does not accept the distinct types you want to pass to it. You can then define a function based on a built-in function, called a sourced user-defined function, that accepts your distinct types. You might need to perform another calculation in your SQL statements for which no built-in function exists. In that situation, you can define and write an SQL function or an external function.

For a detailed discussion of user-defined functions, see “User-defined functions” on page 520.

Creating a distinct type


Distinct types are useful when you want DB2 to handle certain data differently than other data of the same data type. For example, even though all currencies can be declared as type DECIMAL, you do not want euros to be compared to Japanese yen.

To create a distinct type:

Issue the CREATE DISTINCT TYPE statement. For example, you can create distinct types for euros and yen by issuing the following SQL statements:

```
CREATE DISTINCT TYPE EURO AS DECIMAL(9,2);  
CREATE DISTINCT TYPE JAPANESE_YEN AS DECIMAL(9,2);
```

Related reference:

 CREATE TYPE (distinct) (DB2 SQL)

Distinct types

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

Each distinct type has the same internal representation as a built-in data type.

Suppose you want to define some audio and video data in a DB2 table. You can define columns for both types of data as BLOB, but you might want to use a data type that more specifically describes the data. To do that, define distinct types. You can then use those types when you define columns in a table or manipulate the data in those columns. For example, you can define distinct types for the audio and video data like this:

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

Then, your CREATE TABLE statement might look like this:

```
CREATE TABLE VIDEO_CATALOG;
  (VIDEO_NUMBER CHAR(6) NOT NULL,
   VIDEO_SOUND AUDIO,
   VIDEO_PICS VIDEO,
   ROW_ID ROWID NOT NULL GENERATED ALWAYS);
```

For more information on LOB data, see “Large objects (LOBs)” on page 465.

After you define distinct types and columns of those types, you can use those data types in the same way you use built-in types. You can use the data types in assignments, comparisons, function invocations, and stored procedure calls. However, when you assign one column value to another or compare two column values, those values must be of the same distinct type. For example, you must assign a column value of type VIDEO to a column of type VIDEO, and you can compare a column value of type AUDIO only to a column of type AUDIO. When you assign a host variable value to a column with a distinct type, you can use any host data type that is compatible with the source data type of the distinct type. For example, to receive an AUDIO or VIDEO value, you can define a host variable like this:

```
SQL TYPE IS BLOB (1M) HVAV;
```

When you use a distinct type as an argument to a function, a version of that function that accepts that distinct type must exist. For example, if function SIZE takes a BLOB type as input, you cannot automatically use a value of type AUDIO as input. However, you can create a sourced user-defined function that takes the AUDIO type as input. For example:

```
CREATE FUNCTION SIZE(AUDIO)
  RETURNS INTEGER
  SOURCE SIZE(BLOB(1M));
```

Using distinct types in application programs: The main reason to use distinct types is because DB2 enforces *strong typing* for distinct types. Strong typing ensures that only functions, procedures, comparisons, and assignments that are defined for a data type can be used.

For example, if you have defined a user-defined function to convert U.S. dollars to euro currency, you do not want anyone to use this same user-defined function to convert Japanese yen to euros because the U.S. dollars to euros function returns the wrong amount. Suppose you define three distinct types:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL(9,2);
CREATE DISTINCT TYPE EURO AS DECIMAL(9,2);
CREATE DISTINCT TYPE JAPANESE_YEN AS DECIMAL(9,2);
```

If a conversion function is defined that takes an input parameter of type US_DOLLAR as input, DB2 returns an error if you try to execute the function with an input parameter of type JAPANESE_YEN.

Example of distinct types, user-defined functions, and LOBs

You can create and use a distinct type based on a LOB data type.

The example in this topic demonstrates the following concepts:

- Creating a distinct type based on a LOB data type

- Defining a user-defined function with a distinct type as an argument
- Creating a table with a distinct type column that is based on a LOB type
- Defining a LOB table space, auxiliary table, and auxiliary index
- Inserting data from a host variable into a distinct type column based on a LOB column
- Executing a query that contains a user-defined function invocation
- Casting a LOB locator to the input data type of a user-defined function

Suppose that you keep electronic mail documents that are sent to your company in a DB2 table. The DB2 data type of an electronic mail document is a CLOB, but you define it as a distinct type so that you can control the types of operations that are performed on the electronic mail. The distinct type is defined like this:

```
CREATE DISTINCT TYPE E_MAIL AS CLOB(5M);
```

You have also defined and written user-defined functions to search for and return the following information about an electronic mail document:

- Subject
- Sender
- Date sent
- Message content
- Indicator of whether the document contains a user-specified string

The user-defined function definitions look like this:

```
CREATE FUNCTION SUBJECT(E_MAIL)
  RETURNS VARCHAR(200)
  EXTERNAL NAME 'SUBJECT'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION SENDER(E_MAIL)
  RETURNS VARCHAR(200)
  EXTERNAL NAME 'SENDER'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION SENDING_DATE(E_MAIL)
  RETURNS DATE
  EXTERNAL NAME 'SENDDATE'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION CONTENTS(E_MAIL)
  RETURNS CLOB(1M)
  EXTERNAL NAME 'CONTENTS'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION;

CREATE FUNCTION CONTAINS(E_MAIL, VARCHAR (200))
  RETURNS INTEGER
  EXTERNAL NAME 'CONTAINS'
  LANGUAGE C
```

```

PARAMETER STYLE SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION;

```

The table that contains the electronic mail documents is defined like this:

```

CREATE TABLE DOCUMENTS
  (LAST_UPDATE_TIME  TIMESTAMP,
   DOC_ROWID         ROWID NOT NULL GENERATED ALWAYS,
   A_DOCUMENT        E_MAIL);

```

Because the table contains a column with a source data type of CLOB, the table requires an associated LOB table space, auxiliary table, and index on the auxiliary table. Use statements like this to define the LOB table space, the auxiliary table, and the index:

```

CREATE LOB TABLESPACE DOCTSLOB
  LOG YES
  GBPCACHE SYSTEM;

CREATE AUX TABLE DOCAUX_TABLE
  IN DOCTSLOB
  STORES DOCUMENTS COLUMN A_DOCUMENT;

CREATE INDEX A_IX_DOC ON DOCAUX_TABLE;

```

To populate the document table, you write code that executes an INSERT statement to put the first part of a document in the table, and then executes multiple UPDATE statements to concatenate the remaining parts of the document. For example:

```

EXEC SQL BEGIN DECLARE SECTION;
  char hv_current_time[26];
  SQL TYPE IS CLOB (1M) hv_doc;
EXEC SQL END DECLARE SECTION;
/* Determine the current time and put this value */
/* into host variable hv_current_time.          */
/* Read up to 1 MB of document data from a file */
/* into host variable hv_doc.                   */
:
:
/* Insert the time value and the first 1 MB of */
/* document data into the table.                */
EXEC SQL INSERT INTO DOCUMENTS
  VALUES(:hv_current_time, DEFAULT, E_MAIL(:hv_doc));

/* Although there is more document data in the */
/* file, read up to 1 MB more of data, and then */
/* use an UPDATE statement like this one to     */
/* concatenate the data in the host variable    */
/* to the existing data in the table.           */
EXEC SQL UPDATE DOCUMENTS
  SET A_DOCUMENT = A_DOCUMENT || E_MAIL(:hv_doc)
  WHERE LAST_UPDATE_TIME = :hv_current_time;

```

Now that the data is in the table, you can execute queries to learn more about the documents. For example, you can execute this query to determine which documents contain the word "performance":

```

SELECT SENDER(A_DOCUMENT), SENDING_DATE(A_DOCUMENT),
  SUBJECT(A_DOCUMENT)
FROM DOCUMENTS
WHERE CONTAINS(A_DOCUMENT,'performance') = 1;

```

Because the electronic mail documents can be very large, you might want to use LOB locators to manipulate the document data instead of fetching all of a

document into a host variable. You can use a LOB locator on any distinct type that is defined on one of the LOB types. The following example shows how you can cast a LOB locator as a distinct type, and then use the result in a user-defined function that takes a distinct type as an argument:

```
EXEC SQL BEGIN DECLARE SECTION
    long hv_len;
    char hv_subject[200];
    SQL TYPE IS CLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION
:
:
/* Select a document into a CLOB locator.          */
EXEC SQL SELECT A_DOCUMENT, SUBJECT(A_DOCUMENT)
    INTO :hv_email_locator, :hv_subject
    FROM DOCUMENTS
    WHERE LAST_UPDATE_TIME = :hv_current_time;
:
:
/* Extract the subject from the document. The      */
/* SUBJECT function takes an argument of type     */
/* E_MAIL, so cast the CLOB locator as E_MAIL.   */
EXEC SQL SET :hv_subject =
    SUBJECT(CAST(:hv_email_locator AS E_MAIL));
:
:
```

Defining a user-defined function

User-defined functions are small programs that you can write to perform an operation. You can create your own external functions, sourced functions, or SQL functions. You can then use that function wherever you can use a built-in function.

To define a user-defined function:

1. Determine the characteristics of the user-defined function, such as the user-defined function name, schema (qualifier), and number and data types of the input parameters and the types of the values returned.
2. Execute a CREATE FUNCTION statement to register the information in the DB2 catalog.

If you discover after you define the function that any of these characteristics is not appropriate for the function, you can use an ALTER FUNCTION statement to change information in the definition. You cannot use ALTER FUNCTION to change some of the characteristics of a user-defined function definition.

Examples

Example: Definition for an external user-defined scalar function: A programmer develops a user-defined function that searches for a string of maximum length 200 in a CLOB value whose maximum length is 500 KB. This CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
    RETURNS INTEGER
    CAST FROM FLOAT
    SPECIFIC FINDSTRINCLOB
    EXTERNAL NAME 'FINDSTR'
    LANGUAGE C
    PARAMETER STYLE SQL
    NO SQL
    DETERMINISTIC
    NO EXTERNAL ACTION
    FENCED
    STOP AFTER 3 FAILURES;
```


The output from the user-defined function is of type float, but users require integer output for their SQL statements. The user-defined function is written in C and contains no SQL statements. The function is defined to stop when the number of abnormal terminations is equal to 3.

Example: Definition for an external user-defined scalar function that overloads an operator: A programmer has written a user-defined function that overloads the built-in SQL division operator (/). That is, this user-defined function is invoked when an application program executes a statement like either of the following:

```
UPDATE TABLE1 SET INTCOL1=INTCOL2/INTCOL3;  
UPDATE TABLE1 SET INTCOL1="/"(INTCOL2,INTCOL3);
```

The user-defined function takes two integer values as input. The output from the user-defined function is of type integer. The user-defined function is in the MATH schema, is written in assembler, and contains no SQL statements. This CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION MATH."/" (INT, INT)  
  RETURNS INTEGER  
  SPECIFIC DIVIDE  
  EXTERNAL NAME 'DIVIDE'  
  LANGUAGE ASSEMBLE  
  PARAMETER STYLE SQL  
  NO SQL  
  DETERMINISTIC  
  NO EXTERNAL ACTION  
  FENCED;
```

Suppose that you want the FINDSTRING user-defined function to work on BLOB data types, as well as CLOB types. You can define another instance of the user-defined function that specifies a BLOB type as input:

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))  
  RETURNS INTEGER  
  CAST FROM FLOAT  
  SPECIFIC FINDSTRINBLOB  
  EXTERNAL NAME 'FNDBLOB'  
  LANGUAGE C  
  PARAMETER STYLE SQL  
  NO SQL  
  DETERMINISTIC  
  NO EXTERNAL ACTION  
  FENCED  
  STOP AFTER 3 FAILURES;
```

Each instance of FINDSTRING uses a different application program to implement the user-defined function.

Example: Definition for a sourced user-defined function: Suppose you need a user-defined function that finds a string in a value with a distinct type of BOAT. BOAT is based on a BLOB data type. User-defined function FINDSTRING has already been defined. FINDSTRING takes a BLOB data type and performs the required function. The specific name for FINDSTRING is FINDSTRINBLOB.

You can therefore define a sourced user-defined function based on FINDSTRING to do the string search on values of type BOAT. This CREATE FUNCTION statement defines the sourced user-defined function:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))  
  RETURNS INTEGER  
  SPECIFIC FINDSTRINBOAT  
  SOURCE SPECIFIC FINDSTRINBLOB;
```


Example: Definition for an SQL user-defined function: You can define an SQL user-defined function for the tangent of a value by using the existing built-in SIN and COS functions:

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN SIN(X)/COS(X);
```

Example: Definition for an external user-defined table function: An application programmer develops a user-defined function that receives two values and returns a table. The two input values are:

- A character string of maximum length 30 that describes a subject
- A character string of maximum length 255 that contains text to search for

The user-defined function scans documents on the subject for the search string and returns a list of documents that match the search criteria, with an abstract for each document. The list is in the form of a two-column table. The first column is a character column of length 16 that contains document IDs. The second column is a varying-character column of maximum length 5000 that contains document abstracts.

The user-defined function is written in COBOL, uses SQL only to perform queries, always produces the same output for given input, and should not execute as a parallel task. The program is reentrant, and successive invocations of the user-defined function share information. You expect an invocation of the user-defined function to return about 20 rows.

The following CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16), DOC_ABSTRACT VARCHAR(5000))
  EXTERNAL NAME 'DOCMTCH'
  LANGUAGE COBOL
  PARAMETER STYLE SQL
  READS SQL DATA
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20;
```

Related concepts:

 [Sample user-defined functions \(DB2 SQL\)](#)

Related tasks:

 [Controlling user-defined functions \(DB2 Administration Guide\)](#)

Related reference:

“Components of a user-defined function definition” on page 523

 [ALTER FUNCTION \(external\) \(DB2 SQL\)](#)

 [ALTER FUNCTION \(compiled SQL scalar\) \(DB2 SQL\)](#)

 [CREATE FUNCTION \(DB2 SQL\)](#)

User-defined functions

A *user-defined function* is an extension to the SQL language. A user-defined function is a small program that you write, similar to a host language subprogram or function. However, a user-defined function is often the better choice for an SQL application because you can invoke it in an SQL statement.

This section contains information that applies to all user-defined functions and specific information about user-defined functions in languages other than Java.

The types of user-defined functions are:

- *Sourced* user-defined functions, which are based on existing built-in functions or user-defined functions
- *External* user-defined functions, which a programmer writes in a host language
- *SQL* user-defined functions, which contain the source code for the user-defined function in the user-defined function definition

User-defined functions can also be categorized as *user-defined scalar functions* or *user-defined table functions*:

- A user-defined scalar function returns a single-value answer each time it is invoked
- A user-defined table function returns a table to the SQL statement that references it

Creating and using a user-defined function involves these steps:

- Setting up the environment for user-defined functions

A systems administrator probably performs this step. The user-defined function environment is shown in the following figure.

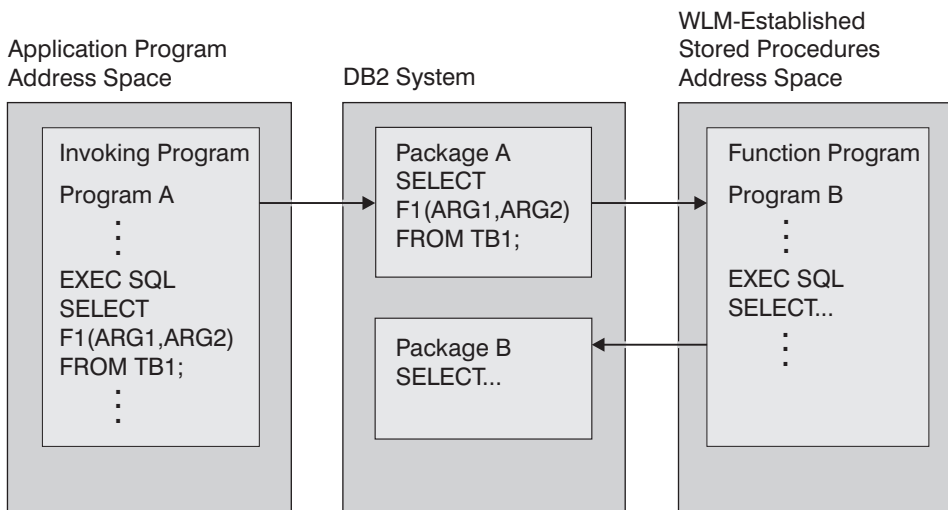


Figure 26. The user-defined function environment

It contains an application address space, from which a program invokes a user-defined function; a DB2 system, where the packages from the user-defined function are run; and a WLM-established address space, where the user-defined function is executed. The steps for setting up and maintaining the user-defined function environment are the same as for setting up and maintaining the environment for stored procedures in WLM-established address spaces.

- Writing and preparing the user-defined function
This step is necessary only for an external user-defined function.
The person who performs this step is called the user-defined function *implementer*.
- Defining the user-defined function to DB2
The person who performs this step is called the user-defined function *definer*.
- Invoking the user-defined function from an SQL application
The person who performs this step is called the user-defined function *invoker*.

Related concepts:

 [Java stored procedures and user-defined functions \(DB2 Application Programming for Java\)](#)

External user-defined functions

An external user-defined function is a function that is written in a programming language. These functions can return a single value or a complete table.

You can write an external user-defined function in assembler, C, C++, COBOL, PL/I, or Java™. User-defined functions that are written in COBOL can include object-oriented extensions, just as other DB2 COBOL programs can. User-defined functions that are written in Java follow coding guidelines and restrictions specific to Java. For information about writing Java user-defined functions, see the topic "Creating Java stored procedures and user-defined functions".

SQL scalar functions

An SQL scalar function is a user-defined function written in SQL and it returns a single value each time it is invoked. There are two kinds of SQL scalar functions, inline and non-inline.

All SQL scalar functions that were created prior to Version 10 are inline SQL scalar function. Beginning with Version 10, SQL scalar functions may be created as either inline or non-inline.

DB2 determines whether an SQL scalar function is inline or non-inline according to whether or not the CREATE FUNCTION statement that defines the function makes use of the enhanced features.

An SQL scalar function that is created without the use of any of the enhanced features for the CREATE FUNCTION statement is considered to be an inline SQL scalar function. All other SQL scalar functions will be considered to be a non-inline SQL scalar function. One exception to this rule is that if the function could have been defined prior to Version 10, except for the XML data type in the input parameters or in the RETURNS parameter, then the function will still be considered to be an inline SQL scalar function.

An inline SQL scalar function has a body with a single RETURN statement. The RETURN statement can return either a NULL value or a simple expression that does not reference a scalar fullselect. No package will be generated for an inline SQL scalar function; during the preparation of an SQL statement that references the function, the expression specified in the RETURN statement of the function is simply inlined into that SQL statement. The versioning of SQL functions and the new features for ALTER FUNCTION statement and enhanced BIND PACKAGE DEPLOY command for non-inline SQL scalar functions do not apply to inline SQL scalar functions.

A non-inline SQL scalar function can have a body with logic written in SQL PL language. It can make use of any of the enhanced features for the CREATE FUNCTION statement including the support for TABLE LOCATOR data type for parameters, various new options, and enhanced RETURN statement that allows reference to a scalar fullselect. A package is created for a non-inline SQL scalar function. The versioning of SQL functions, new features for ALTER FUNCTION statement, and enhanced BIND PACKAGE DEPLOY command do apply to non-inline SQL scalar functions. Once the first version of an SQL function has been created as non-inline, any subsequent version added or replaced for the function will also be non-inline.

Non-inline SQL scalar functions include the following support for versioning and source code management:

- Define multiple versions of an SQL scalar function, where one version is considered the “active” version
- Activate a particular version of an SQL scalar function
- Alter the routine options that are associated with a version of an SQL scalar function
- Define a new version of an SQL scalar function by specifying the same function signature as the current version, and different routine options and function body
- Replace the definition of an existing version by specifying the same function signature as the current version, and different routine options and function body
- Drop a version of an SQL scalar function.
- Fall back to a previous version without requiring an explicit rebind or recompile

You can deploy non-inline SQL scalar functions to multiple servers to allow a wider community to use functions that have been thoroughly tested, without the risk of changing the logic in the routine body. Use the Unified Debugger to remotely debug non-inline SQL scalar functions that execute on DB2 for z/OS servers.

To prepare an SQL scalar function for execution, you execute the CREATE FUNCTION statement, either statically or dynamically.

Related tasks:

“Defining a user-defined function” on page 517

Related reference:

 CREATE FUNCTION (compiled SQL scalar) (DB2 SQL)

SQL table functions

An SQL table function is a function that is written exclusively in SQL statements and returns a single result table.

An SQL table function can define a parameter as a distinct type, define a parameter for a transition table (for example, the TABLE LIKE ... AS LOCATOR syntax), and include a single SQL PL RETURN statement that returns a result table

The CREATE statement for an SQL table function is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

The ALTER statement for an SQL table function can be embedded in an application program or issued interactively. The ALTER statement is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Sourced functions

A *sourced function* is a function that invokes another function that already exists at the server. The function inherits the attributes of the underlying source function. The source function can be built-in, external, SQL, or sourced.

Use sourced functions to build upon existing built-in functions or other user-defined functions. Sourced functions are useful to extend built-in aggregate and scalar functions for use on distinct types.

To implement a sourced function, issue a CREATE FUNCTION statement with the name of the function upon which you want to base the sourced function.

Related reference:

 CREATE FUNCTION (sourced) (DB2 SQL)

Components of a user-defined function definition

The characteristics that you specify for a user-defined function depend on whether the function is sourced, external, or SQL. You specify these characteristics in a CREATE FUNCTION or ALTER FUNCTION statement.

The following table lists the characteristics of a user-defined function, the corresponding parameters in the CREATE FUNCTION and ALTER FUNCTION statements, and which parameters are valid for sourced, external, and SQL user-defined functions.

Table 85. Characteristics of a user-defined function

Characteristic	CREATE FUNCTION or ALTER FUNCTION option	Valid in sourced function?	Valid in external function?	Valid in SQL function?
User-defined function name	none	Yes	Yes	Yes
Input parameter types and encoding schemes	none	Yes	Yes	Yes
Output parameter types and encoding schemes	RETURNS RETURNS TABLE ¹	Yes	Yes	Yes ²
Specific name	SPECIFIC	Yes	Yes	Yes
External name	EXTERNAL NAME	No	Yes	No
Language	LANGUAGE ASSEMBLE LANGUAGE C LANGUAGE COBOL LANGUAGE PLI LANGUAGE JAVA LANGUAGE SQL	No	Yes ³	Yes ⁴
Deterministic or non-deterministic	NOT DETERMINISTIC DETERMINISTIC	No	Yes	Yes
Types of SQL statements in the function	NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA	No	Yes ⁵	Yes ⁶

Table 85. Characteristics of a user-defined function (continued)

Characteristic	CREATE FUNCTION or ALTER FUNCTION option	Valid in sourced function?	Valid in external function?	Valid in SQL function?
Name of source function	SOURCE	Yes	No	No
Parameter style	PARAMETER STYLE SQL PARAMETER STYLE JAVA	No	Yes ⁷	No
Address space for user-defined functions	FENCED	No	Yes	No
Call with null input	RETURNS NULL ON NULL INPUT CALLED ON NULL INPUT	No	Yes	Yes ⁸
External actions	EXTERNAL ACTION NO EXTERNAL ACTION	No	Yes	Yes
Scratchpad specification	NO SCRATCHPAD SCRATCHPAD <i>length</i>	No	Yes	No
Call function after SQL processing	NO FINAL CALL FINAL CALL	No	Yes	No
Consider function for parallel processing	ALLOW PARALLEL DISALLOW PARALLEL	No	Yes ⁵	No
Package collection	NO COLLID COLLID <i>collection-id</i>	No	Yes	No
WLM environment	WLM ENVIRONMENT <i>name</i> WLM ENVIRONMENT <i>name</i> ,*	No	Yes	No
CPU time for a function invocation	ASUTIME NO LIMIT ASUTIME LIMIT <i>integer</i>	No	Yes	No
Load module stays in memory	STAY RESIDENT NO STAY RESIDENT YES	No	Yes	No
Program type	PROGRAM TYPE MAIN PROGRAM TYPE SUB	No	Yes	No
Security	SECURITY DB2 SECURITY USER SECURITY DEFINER	No	Yes	No
run time options	RUN OPTIONS <i>options</i>	No	Yes	No
Pass DB2 environment information	NO DBINFO DBINFO	No	Yes	No
Expected number of rows returned	CARDINALITY <i>integer</i>	No	Yes ¹	No
Function resolution is based on the declared parameter types	STATIC DISPATCH	No	No	Yes
SQL expression that evaluates to the value returned by the function	none	No	No	Yes
Encoding scheme for all string parameters	PARAMETER CCSID EBCDIC PARAMETER CCSID ASCII PARAMETER CCSID UNICODE	No	Yes	Yes

Table 85. Characteristics of a user-defined function (continued)

Characteristic	CREATE FUNCTION or ALTER FUNCTION option	Valid in sourced function?	Valid in external function?	Valid in SQL function?
For functions that are defined as LANGUAGE C, the representation of VARCHAR parameters and, if applicable, the returned result.	PARAMETER VARCHAR NULTERM PARAMETER VARCHAR STRUCTURE ⁹	No	Yes	No
Number of abnormal terminations before the function is stopped	STOP AFTER SYSTEM DEFAULT FAILURES STOP AFTER <i>n</i> FAILURES CONTINUE AFTER FAILURE	No	Yes	No
Identifies the list of package collections that is to be used when the stored procedure is executed.	PACKAGE PATH <i>package-path</i> NO PACKAGE PATH	No	Yes	No

Notes:

1. RETURNS TABLE and CARDINALITY are valid only for user-defined table functions. For a single query, you can override the CARDINALITY value by specifying a CARDINALITY clause for the invocation of a user-defined table function in the SELECT statement.
2. An SQL user-defined function can return only one scalar value.
3. LANGUAGE SQL is not valid for an external user-defined function.
4. Only LANGUAGE SQL is valid for an SQL user-defined function.
5. MODIFIES SQL DATA and ALLOW PARALLEL are not valid for user-defined table functions.
6. MODIFIES SQL DATA and NO SQL are not valid for SQL user-defined functions.
7. PARAMETER STYLE JAVA is valid only with LANGUAGE JAVA. PARAMETER STYLE SQL is valid only with LANGUAGE values other than LANGUAGE JAVA.
8. RETURNS NULL ON NULL INPUT is not valid for an SQL user-defined function.
9. The PARAMETER VARCHAR clause can be specified in CREATE FUNCTION statements only.

Related reference:

 CREATE FUNCTION (DB2 SQL)

Writing an external user-defined function

An *external user-defined function* is written in a programming language and is similar to other SQL programs. You can include static or dynamic SQL statements, IFI calls, and DB2 commands that are issued through IFI calls.

Your user-defined function can also access remote data using the following methods:

- DRDA access using three-part names or aliases for three-part names
- DRDA access using CONNECT or SET CONNECTION statements

You can write an external user-defined function in assembler, C, C++, COBOL, PL/I, or Java. User-defined functions that are written in COBOL can include object-oriented extensions, just as other DB2 COBOL programs can. User-defined functions that are written in Java follow coding guidelines and restrictions specific to Java.

Restrictions on user-defined function programs: Observe these restrictions when you write a user-defined function:

- Because DB2 uses the Resource Recovery Services attachment facility (RRSAF) as its interface with your user-defined function, you must not include RRSAF calls in your user-defined function. DB2 rejects any RRSAF calls that it finds in a user-defined function.
- If your user-defined function is not defined with parameters `SCRATCHPAD` or `EXTERNAL ACTION`, the user-defined function is not guaranteed to execute under the same task each time it is invoked.
- You cannot execute `COMMIT` or `ROLLBACK` statements in your user-defined function.
- You must close all cursors that were opened within a user-defined scalar function. DB2 returns an SQL error if a user-defined scalar function does not close all cursors that it opened before it completes.
- When you choose the language in which to write a user-defined function program, be aware of restrictions on the number of parameters that can be passed to a routine in that language. User-defined table functions in particular can require large numbers of parameters. Consult the programming guide for the language in which you plan to write the user-defined function for information about the number of parameters that can be passed.
- You cannot pass LOB file reference variables as parameters to user-defined functions.
- User-defined functions cannot return LOB file reference variables.
- You cannot pass parameters with the type `XML` to user-defined functions. You can specify tables or views that contain XML columns as table locator parameters. However, you cannot reference the XML columns in the body of the user-defined function.

Coding your user-defined function as a main program or as a subprogram: You can code your user-defined function as either a main program or a subprogram. The way that you code your program must agree with the way you defined the user-defined function: with the `PROGRAM TYPE MAIN` or `PROGRAM TYPE SUB` parameter. The main difference is that when a main program starts, Language Environment allocates the application program storage that the external user-defined function uses. When a main program ends, Language Environment closes files and releases dynamically allocated storage.

If you code your user-defined function as a subprogram and manage the storage and files yourself, you can get better performance. The user-defined function should always free any allocated storage before it exits. To keep data between invocations of the user-defined function, use a scratchpad.

You must code a user-defined table function that accesses external resources as a subprogram. Also ensure that the definer specifies the `EXTERNAL ACTION` parameter in the `CREATE FUNCTION` or `ALTER FUNCTION` statement. Program variables for a subprogram persist between invocations of the user-defined function, and use of the `EXTERNAL ACTION` parameter ensures that the user-defined function stays in the same address space from one invocation to another.

Parallelism considerations: If the definer specifies the parameter `ALLOW PARALLEL` in the definition of a user-defined scalar function, and the invoking SQL statement runs in parallel, the function can run under a parallel task. DB2 executes a separate instance of the user-defined function for each parallel task.

When you write your function program, you need to understand how the following parameter values interact with ALLOW PARALLEL so that you can avoid unexpected results:

- SCRATCHPAD

When an SQL statement invokes a user-defined function that is defined with the ALLOW PARALLEL parameter, DB2 allocates one scratchpad for each parallel task of each reference to the function. This can lead to unpredictable or incorrect results.

For example, suppose that the user-defined function uses the scratchpad to count the number of times it is invoked. If a scratchpad is allocated for each parallel task, this count is the number of invocations done by the *parallel task* and not for the entire SQL statement, which is not the result that is wanted.

- FINAL CALL

If a user-defined function performs an external action, such as sending a note, for each final call to the function, one note is sent for each parallel task instead of once for the function invocation.

- EXTERNAL ACTION

Some user-defined functions with external actions can receive incorrect results if the function is executed by parallel tasks.

For example, if the function sends a note for each initial call to the function, one note is sent for each parallel task instead of once for the function invocation.

- NOT DETERMINISTIC

A user-defined function that is non-deterministic can generate incorrect results if it is run under a parallel task.

For example, suppose that you execute the following query under parallel tasks:
SELECT * FROM T1 WHERE C1 = COUNTER();

COUNTER is a user-defined function that increments a variable in the scratchpad every time it is invoked. Counter is non-deterministic because the same input does not always produce the same output. Table T1 contains one column, C1, that has the following values:

1
2
3
4
5
6
7
8
9
10

When the query is executed with no parallelism, DB2 invokes COUNTER once for each row of table T1, and there is one scratchpad for counter, which DB2 initializes the first time that COUNTER executes. COUNTER returns 1 the first time it executes, 2 the second time, and so on. The result table for the query has the following values:

1
2
3
4
5
6

7
8
9
10

Now suppose that the query is run with parallelism, and DB2 creates three parallel tasks. DB2 executes the predicate `WHERE C1 = COUNTER()` for each parallel task. This means that each parallel task invokes its own instance of the user-defined function and has its own scratchpad. DB2 initializes the scratchpad to zero on the first call to the user-defined function for each parallel task.

If parallel task 1 processes rows 1 to 3, parallel task 2 processes rows 4 to 6, and parallel task 3 processes rows 7 to 10, the following results occur:

- When parallel task 1 executes, C1 has values 1, 2, and 3, and `COUNTER` returns values 1, 2, and 3, so the query returns values 1, 2, and 3.
- When parallel task 2 executes, C1 has values 4, 5, and 6, but `COUNTER` returns values 1, 2, and 3, so the query returns no rows.
- When parallel task 3, executes, C1 has values 7, 8, 9, and 10, but `COUNTER` returns values 1, 2, 3, and 4, so the query returns no rows.

Thus, instead of returning the 10 rows that you might expect from the query, DB2 returns only 3 rows.

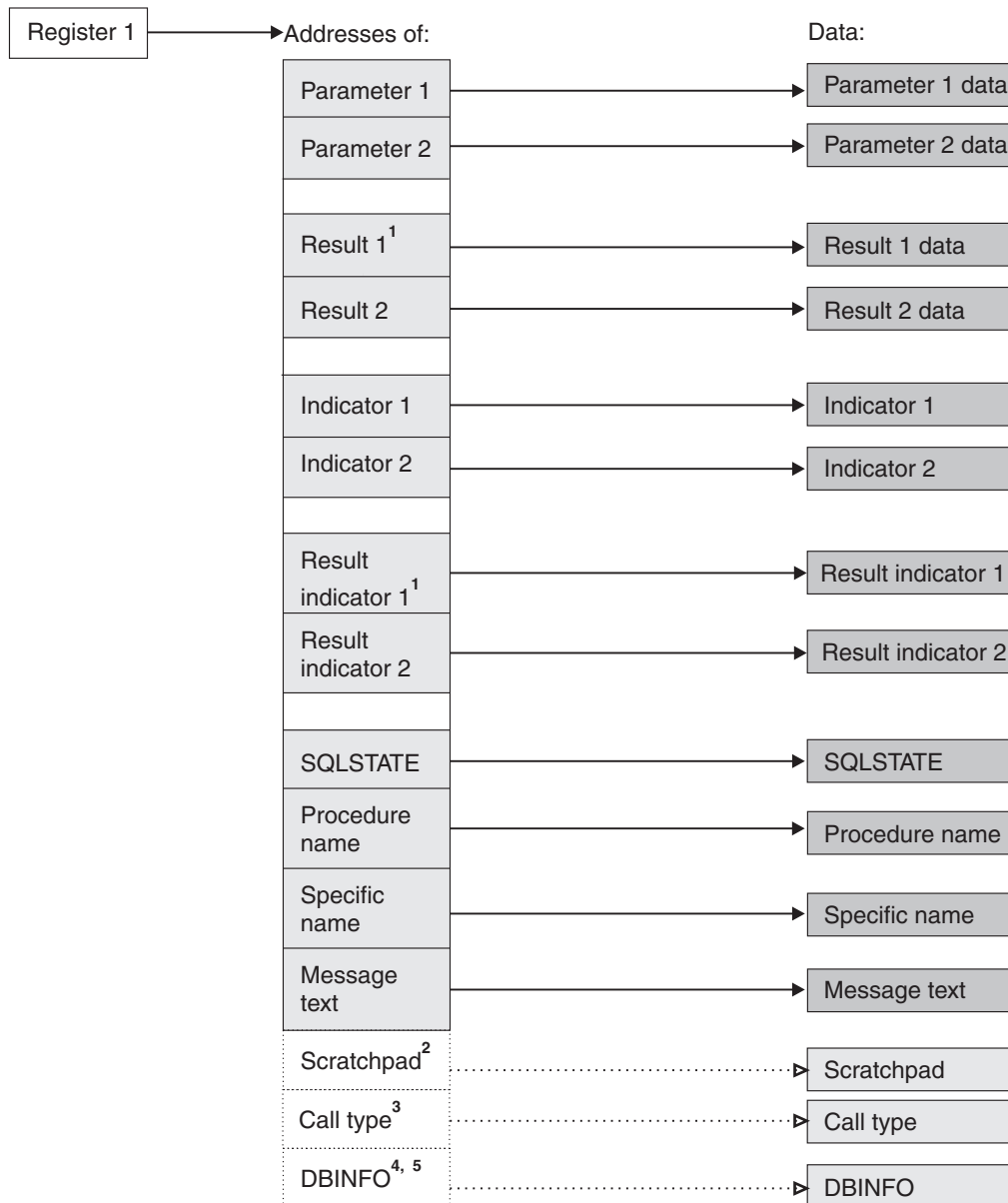
Related concepts:

 [Java stored procedures and user-defined functions \(DB2 Application Programming for Java\)](#)

Parameters for external user-defined functions

To receive parameters from and pass parameters to an invoker of an external user-defined function, you must understand the structure of the parameter list. You must also understand the meaning of each parameter, and whether DB2 or your user-defined function sets the value of each parameter.

The following figure shows the structure of the parameter list that DB2 passes to a user-defined function. An explanation of each parameter follows.



1. For a user-defined scalar function, only one result and one result indicator are passed.
2. Passed if the SCRATCHPAD option is specified in the user-defined function definition.
3. Passed if the FINAL CALL option is specified in a user-defined scalar function definition; always passed for a user-defined table function.
4. For PL/I, this value is the address of a pointer to the DBINFO data.
5. Passed if the DBINFO option is specified in the user-defined function definition.

Figure 27. Parameter conventions for a user-defined function

Input parameter values

DB2 obtains the input parameters from the invoker's parameter list, and your user-defined function receives those parameters according to the rules of the host language in which the user-defined function is written. The number of input parameters is the same as the number of parameters in the user-defined function invocation. If one of the parameters in the function invocation is an expression, DB2 evaluates the expression and assigns the result of the expression to the parameter.

For all data types except LOBs, ROWIDs, locators, and VARCHAR (with C language), see the tables listed in the following table for the host data types that are compatible with the data types in the user-defined function definition.

Table 86. Listing of tables of compatible data types

Language	Compatible data types table
Assembler	"Compatibility of SQL and language data types" on page 180
C	"Compatibility of SQL and language data types" on page 180
COBOL	"Compatibility of SQL and language data types" on page 180
PL/I	"Compatibility of SQL and language data types" on page 180

For LOBs, ROWIDs, and locators, see the following table for the assembler data types that are compatible with the data types in the user-defined function definition.

Table 87. Compatible assembler language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	Assembler declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	DS FL4
BLOB(<i>n</i>)	If $n \leq 65535$: var DS 0FL4 var_length DS FL4 var_data DS CLn If $n > 65535$: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+($n-65535$)
CLOB(<i>n</i>)	If $n \leq 65535$: var DS 0FL4 var_length DS FL4 var_data DS CLn If $n > 65535$: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+($n-65535$)
DBCLOB(<i>n</i>)	If $n (=2*n) \leq 65534$: var DS 0FL4 var_length DS FL4 var_data DS CLm If $n > 65534$: var DS 0FL4 var_length DS FL4 var_data DS CL65534 ORG var_data+($m-65534$)

Table 87. Compatible assembler language declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	Assembler declaration
ROWID	DS HL2,CL40

For LOBs, ROWIDs, VARCHARs, and locators see the following table for the C data types that are compatible with the data types in the user-defined function definition.

Table 88. Compatible C language declarations for LOBs, ROWIDs, VARCHARs, and locators

SQL data type in definition ¹	C declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	unsigned long
BLOB(<i>n</i>)	struct {unsigned long length; char data[<i>n</i>]; } var;
CLOB(<i>n</i>)	struct {unsigned long length; char var_data[<i>n</i>]; } var;
DBCLOB(<i>n</i>)	struct {unsigned long length; sqldbchar data[<i>n</i>]; } var;
ROWID	struct { short int length; char data[40]; } var;
VARCHAR(<i>n</i>) ²	If PARAMETER VARCHAR NULTERM is specified or implied: char data[<i>n</i> +1]; If PARAMETER VARCHAR STRUCTURE is specified: struct {short len; char data[<i>n</i>]; } var;

Note:

1. The SQLUDF file, which is in data set DSNA10.SDSNC.H, includes the typedef sqldbchar. Using sqldbchar lets you manipulate DBCS and Unicode UTF-16 data in the same format in which it is stored in DB2. sqldbchar also makes applications easier to port to other DB2 platforms.
2. This row does not apply to VARCHAR(*n*) FOR BIT DATA. BIT DATA is always passed in a structured representation.

For LOBs, ROWIDs, and locators, see the following table for the COBOL data types that are compatible with the data types in the user-defined function definition.

Table 89. Compatible COBOL declarations for LOBs, ROWIDs, and locators

SQL data type in definition	COBOL declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	01 var PIC S9(9) COMP-5
BLOB(<i>n</i>)	01 var. 49 var-LENGTH PIC S9(9) COMP-5. 49 var-DATA PIC X(<i>n</i>).
CLOB(<i>n</i>)	01 var. 49 var-LENGTH PIC S9(9) COMP-5. 49 var-DATA PIC X(<i>n</i>).
DBCLOB(<i>n</i>)	01 var. 49 var-LENGTH PIC S9(9) COMP-5. 49 var-DATA PIC G(<i>n</i>) DISPLAY-1.
ROWID	01 var. 49 var-LEN PIC S9(4) COMP-5. 49 var-TEXT PIC X(40).

For LOBs, ROWIDs, and locators, see the following table for the PL/I data types that are compatible with the data types in the user-defined function definition.

Table 90. Compatible PL/I declarations for LOBs, ROWIDs, and locators

SQL data type in definition	PL/I
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	BIN FIXED(31)
BLOB(<i>n</i>)	If <i>n</i> <= 32767: 01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA CHAR(<i>n</i>); If <i>n</i> > 32767: 01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(<i>n</i>) CHAR(32767), 03 var_DATA2 CHAR(mod(<i>n</i> ,32767));

Table 90. Compatible PL/I declarations for LOBs, ROWIDs, and locators (continued)

SQL data type in definition	PL/I
CLOB(<i>n</i>)	<pre> If n <= 32767: 01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA CHAR(<i>n</i>); If n > 32767: 01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(<i>n</i>) CHAR(32767), 03 var_DATA2 CHAR(mod(<i>n</i>,32767)); </pre>
DBCLOB(<i>n</i>)	<pre> If n <= 16383: 01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA GRAPHIC(<i>n</i>); If n > 16383: 01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(<i>n</i>) GRAPHIC(16383), 03 var_DATA2 GRAPHIC(mod(<i>n</i>,16383)); </pre>
ROWID	CHAR(40) VAR;

Result parameters: Set these values in your user-defined function before exiting. For a user-defined scalar function, you return one result parameter. For a user-defined table function, you return the same number of parameters as columns in the RETURNS TABLE clause of the CREATE FUNCTION statement. DB2 allocates a buffer for each result parameter value and passes the buffer address to the user-defined function. Your user-defined function places each result parameter value in its buffer. You must ensure that the length of the value you place in each output buffer does not exceed the buffer length. Use the SQL data type and length in the CREATE FUNCTION statement to determine the buffer length.

See “Parameters for external user-defined functions” on page 528 to determine the host data type to use for each result parameter value. If the CREATE FUNCTION statement contains a CAST FROM clause, use a data type that corresponds to the SQL data type in the CAST FROM clause. Otherwise, use a data type that corresponds to the SQL data type in the RETURNS or RETURNS TABLE clause.

To improve performance for user-defined table functions that return many columns, you can pass values for a subset of columns to the invoker. For example, a user-defined table function might be defined to return 100 columns, but the invoker needs values for only two columns. Use the DBINFO parameter to indicate

to DB2 the columns for which you will return values. Then return values for only those columns. See DBINFO for information about how to indicate the columns of interest.

Input parameter indicators: These are SMALLINT values, which DB2 sets before it passes control to the user-defined function. You use the indicators to determine whether the corresponding input parameters are null. The number and order of the indicators are the same as the number and order of the input parameters. On entry to the user-defined function, each indicator contains one of these values:

0 The input parameter value is not null.

negative

 The input parameter value is null.

Code the user-defined function to check all indicators for null values unless the user-defined function is defined with RETURNS NULL ON NULL INPUT. A user-defined function defined with RETURNS NULL ON NULL INPUT executes only if all input parameters are not null.

Result indicators: These are SMALLINT values, which you must set before the user-defined function ends to indicate to the invoking program whether each result parameter value is null. A user-defined scalar function has one result indicator. A user-defined table function has the same number of result indicators as the number of result parameters. The order of the result indicators is the same as the order of the result parameters. Set each result indicator to one of these values:

0 or positive

 The result parameter is not null.

negative

 The result parameter is null.

SQLSTATE value: This CHAR(5) value represents the SQLSTATE that is passed in to the program from the database manager. The initial value is set to '00000'. Although the SQLSTATE is usually not set by the program, it can be set as the result SQLSTATE that is used to return an error or a warning. Returned values that start with anything other than '00', '01', or '02' are error conditions.

User-defined function name: DB2 sets this value in the parameter list before the user-defined function executes. This value is VARCHAR(257): 128 bytes for the schema name, 1 byte for a period, and 128 bytes for the user-defined function name. If you use the same code to implement multiple versions of a user-defined function, you can use this parameter to determine which version of the function the invoker wants to execute.

Specific name: DB2 sets this value in the parameter list before the user-defined function executes. This value is VARCHAR(128) and is either the specific name from the CREATE FUNCTION statement or a specific name that DB2 generated. If you use the same code to implement multiple versions of a user-defined function, you can use this parameter to determine which version of the function the invoker wants to execute.

Diagnostic message: Your user-defined function can set this CHAR or VARCHAR value to a character string of up to 1000 bytes before exiting. Use this area to pass descriptive information about an error or warning to the invoker.

DB2 allocates a buffer for this area and passes you the buffer address in the parameter list. At least the first 17 bytes of the value you put in the buffer appear in the SQLERRMC field of the SQLCA that is returned to the invoker. The exact number of bytes depends on the number of other tokens in SQLERRMC. Do not use X'FF' in your diagnostic message. DB2 uses this value to delimit tokens.

Scratchpad: If the definer specified SCRATCHPAD in the CREATE FUNCTION statement, DB2 allocates a buffer for the scratchpad area and passes its address to the user-defined function. Before the user-defined function is invoked for the first time in an SQL statement, DB2 sets the length of the scratchpad in the first 4 bytes of the buffer and then sets the scratchpad area to X'00'. DB2 does not reinitialize the scratchpad between invocations of a correlated subquery.

You must ensure that your user-defined function does not write more bytes to the scratchpad than the scratchpad length.

Call type: For a user-defined scalar function, if the definer specified FINAL CALL in the CREATE FUNCTION statement, DB2 passes this parameter to the user-defined function. For a user-defined table function, DB2 always passes this parameter to the user-defined function.

On entry to a *user-defined scalar function*, the call type parameter has one of the following values:

- 1 This is the *first call* to the user-defined function for the SQL statement. For a first call, all input parameters are passed to the user-defined function. In addition, the scratchpad, if allocated, is set to binary zeros.
- 0 This is a *normal call*. For a normal call, all the input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.
- 1 This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

This type of final call occurs when the invoking application explicitly closes a cursor. When a value of 1 is passed to a user-defined function, the user-defined function can execute SQL statements.

- 255 This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because DB2 might have already closed cursors before the final call.

During the first call, your user-defined scalar function should acquire any system resources it needs. During the final call, the user-defined scalar function should release any resources it acquired during the first call. The user-defined scalar function should return a result value only during normal calls. DB2 ignores any results that are returned during a final call. However, the user-defined scalar function can set the SQLSTATE and diagnostic message area during the final call.

If an invoking SQL statement contains more than one user-defined scalar function, and one of those user-defined functions returns an error SQLSTATE, DB2 invokes all of the user-defined functions for a final call, and the invoking SQL statement receives the SQLSTATE of the first user-defined function with an error.

On entry to a *user-defined table function*, the call type parameter has one of the following values:

- 2 This is the *first call* to the user-defined function for the SQL statement. A first call occurs only if the FINAL CALL keyword is specified in the user-defined function definition. For a first call, all input parameters are passed to the user-defined function. In addition, the scratchpad, if allocated, is set to binary zeros.
- 1 This is the *open call* to the user-defined function by an SQL statement. If FINAL CALL is not specified in the user-defined function definition, all input parameters are passed to the user-defined function, and the scratchpad, if allocated, is set to binary zeros during the open call. If FINAL CALL is specified for the user-defined function, DB2 does not modify the scratchpad.
- 0 This is a *fetch call* to the user-defined function by an SQL statement. For a fetch call, all input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.
- 1 This is a *close call*. For a close call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.
- 2 This is a *final call*. This type of final call occurs only if FINAL CALL is specified in the user-defined function definition. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

This type of final call occurs when the invoking application executes a CLOSE CURSOR statement.
- 255 This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, DB2 does not modify it.

This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because DB2 might have already closed cursors before the final call.

If a user-defined table function is defined with FINAL CALL, the user-defined function should allocate any resources it needs during the first call and release those resources during the final call that sets a value of 2.

If a user-defined table function is defined with NO FINAL CALL, the user-defined function should allocate any resources it needs during the open call and release those resources during the close call.

During a fetch call, the user-defined table function should return a row. If the user-defined function has no more rows to return, it should set the SQLSTATE to 02000.

During the close call, a user-defined table function can set the SQLSTATE and diagnostic message area.

If a user-defined table function is invoked from a subquery, the user-defined table function receives a CLOSE call for each invocation of the subquery within the higher level query, and a subsequent OPEN call for the next invocation of the subquery within the higher level query.

DBINFO: If the definer specified DBINFO in the CREATE FUNCTION statement, DB2 passes the DBINFO structure to the user-defined function. DBINFO contains information about the environment of the user-defined function caller. It contains the following fields, in the order shown:

Location name length

An unsigned 2-byte integer field. It contains the length of the location name in the next field.

Location name

A 128-byte character field. It contains the name of the location to which the invoker is currently connected.

Authorization ID length

An unsigned 2-byte integer field. It contains the length of the authorization ID in the next field.

Authorization ID

A 128-byte character field. It contains the authorization ID of the application from which the user-defined function is invoked, padded on the right with blanks. If this user-defined function is nested within other user-defined functions, this value is the authorization ID of the application that invoked the highest-level user-defined function.

Subsystem code page

A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs of the subsystem from which the user-defined function is invoked.

Table qualifier length

An unsigned 2-byte integer field. It contains the length of the table qualifier in the next field. If the table name field is not used, this field contains 0.

Table qualifier

A 128-byte character field. It contains the qualifier of the table that is specified in the table name field.

Table name length

An unsigned 2-byte integer field. It contains the length of the table name in the next field. If the table name field is not used, this field contains 0.

Table name

A 128-byte character field. This field contains the name of the table for the update or insert operation if the reference to the user-defined function in the invoking SQL statement is in one of the following places:

- The right side of a SET clause in an update operation
- In the VALUES list of an insert operation

Otherwise, this field is blank.

Column name length

An unsigned 2-byte integer field. It contains the length of the column name in the next field. If no column name is passed to the user-defined function, this field contains 0.

Column name

A 128-byte character field. This field contains the name of the column that the update or insert operation modifies if the reference to the user-defined function in the invoking SQL statement is in one of the following places:

- The right side of a SET clause in an update operation
- In the VALUES list of an insert operation

Otherwise, this field is blank.

Product information

An 8-byte character field that identifies the product on which the user-defined function executes. This field has the form *pppvrrm*, where:

- *ppp* is a 3-byte product code:

ARI DB2 Server for VSE & VM

DSN DB2 for z/OS

QSQ DB2 for i

SQL DB2 for Linux, UNIX, and Windows

- *vv* is a 2-digit version identifier.
- *rr* is a 2-digit release identifier.
- *m* is a 1-digit maintenance level identifier.

Reserved area

2 bytes.

Operating system

A 4-byte integer field. It identifies the operating system on which the program that invokes the user-defined function 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	z/OS
13	Siemens Nixdorf
15	Windows 95
16	SCO UNIX
18	Linux
19	DYNIX/ptx [®]
24	Linux for S/390 [®]
25	Linux for System z [®]

26	Linux/IA64
27	Linux/PPC
28	Linux/PPC64
29	Linux/AMD64
400	iSeries®

Number of entries in table function column list

An unsigned 2-byte integer field.

Reserved area

26 bytes.

Table function column list pointer

If a table function is defined, this field is a pointer to an array that contains 1000 2-byte integers. DB2 dynamically allocates the array. If a table function is not defined, this pointer is null.

Only the first *n* entries, where *n* is the value in the field entitled number of entries in table function column list, are of interest. *n* is greater than or equal to 0 and less than or equal to the number result columns defined for the user-defined function in the RETURNS TABLE clause of the CREATE FUNCTION statement. The values correspond to the numbers of the columns that the invoking statement needs from the table function. A value of 1 means the first defined result column, 2 means the second defined result column, and so on. The values can be in any order. If *n* is equal to 0, the first array element is 0. This is the case for a statement like the following one, where the invoking statement needs no column values.

```
SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ
```

This array represents an opportunity for optimization. The user-defined function does not need to return all values for all the result columns of the table function. Instead, the user-defined function can return only those columns that are needed in the particular context, which you identify by number in the array. However, if this optimization complicates the user-defined function logic enough to cancel the performance benefit, you might choose to return every defined column.

Unique application identifier

This field is a pointer to a string that uniquely identifies the application's connection to DB2. The string is regenerated for each connection to DB2.

The string is the LUWID, which consists of a fully-qualified LU network name followed by a period and an LUW instance number. The LU network name consists of a 1- to 8-character network ID, a period, and a 1- to 8-character network LU name. The LUW instance number consists of 12 hexadecimal characters that uniquely identify the unit of work.

Reserved area

20 bytes.

If you write your user-defined function in C or C++, you can use the declarations in member SQLUDF of DSNA10.SDSNC.H for many of the passed parameters. To include SQLUDF, make these changes to your program:

- Put this statement in your source code:

```
#include <sqludf.h>
```
- Include the DSNA10.SDSNC.H data set in the SYSLIB concatenation for the compiler step of your program preparation job.

- Specify the NOMARGINS and NOSEQUENCE options in the compiler step of your program preparation job.

Examples of receiving parameters in a user-defined function:

The following examples show how a user-defined function that is written in each of the supported host languages receives the parameter list that is passed by DB2.

These examples assume that the user-defined function is defined with the SCRATCHPAD, FINAL CALL, and DBINFO parameters.

Assembler: The follow figure shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result. For an assembler language user-defined function that is a subprogram, the conventions are the same. In either case, you must include the CEEENTRY and CEEEXIT macros.

```

MYMAIN  CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
        USING PROGAREA,R13

        L    R7,0(R1)          GET POINTER TO PARM1
        MVC  PARM1(4),0(R7)    MOVE VALUE INTO LOCAL COPY OF PARM1
        L    R7,4(R1)          GET POINTER TO PARM2
        MVC  PARM2(4),0(R7)    MOVE VALUE INTO LOCAL COPY OF PARM2
        L    R7,12(R1)         GET POINTER TO INDICATOR 1
        MVC  F_IND1(2),0(R7)   MOVE PARM1 INDICATOR TO LOCAL STORAGE
        LH   R7,F_IND1         MOVE PARM1 INDICATOR INTO R7
        LTR  R7,R7              CHECK IF IT IS NEGATIVE
        BM   NULLIN            IF SO, PARM1 IS NULL
        L    R7,16(R1)         GET POINTER TO INDICATOR 2
        MVC  F_IND2(2),0(R7)   MOVE PARM2 INDICATOR TO LOCAL STORAGE
        LH   R7,F_IND2         MOVE PARM2 INDICATOR INTO R7
        LTR  R7,R7              CHECK IF IT IS NEGATIVE
        BM   NULLIN            IF SO, PARM2 IS NULL

        :
        :
NULLIN  L    R7,8(R1)          GET ADDRESS OF AREA FOR RESULT
        MVC  0(9,R7),RESULT    MOVE A VALUE INTO RESULT AREA
        L    R7,20(R1)         GET ADDRESS OF AREA FOR RESULT IND
        MVC  0(2,R7),=H'0'     MOVE A VALUE INTO INDICATOR AREA

        :
        :
        CEETERM RC=0
*****
*  VARIABLE DECLARATIONS AND EQUATES  *
*****
R1      EQU  1                REGISTER 1
R7      EQU  7                REGISTER 7
PPA     CEEPPA ,              CONSTANTS DESCRIBING THE CODE BLOCK
        LTORG ,               PLACE LITERAL POOL HERE
PROGAREA DSECT
        ORG  ++CEEDSASZ       LEAVE SPACE FOR DSA FIXED PART
PARM1   DS   F                PARAMETER 1
PARM2   DS   F                PARAMETER 2
RESULT  DS   CL9              RESULT
F_IND1  DS   H                INDICATOR FOR PARAMETER 1
F_IND2  DS   H                INDICATOR FOR PARAMETER 2
F_INDR  DS   H                INDICATOR FOR RESULT

PROGSIZE EQU  *-PROGAREA
        CEEDSA ,              MAPPING OF THE DYNAMIC SAVE AREA
        CEECAA ,              MAPPING OF THE COMMON ANCHOR AREA
        END  MYMAIN

```

C or C++: For C or C++ user-defined functions, the conventions for passing parameters are different for main programs and subprograms.

For subprograms, you pass the parameters directly. For main programs, you use the standard `argc` and `argv` variables to access the input and output parameters:

- The `argv` variable contains an array of pointers to the parameters that are passed to the user-defined function. All string parameters that are passed back to DB2 must be null terminated.
 - `argv[0]` contains the address of the load module name for the user-defined function.
 - `argv[1]` through `argv[n]` contain the addresses of parameters 1 through *n*.
- The `argc` variable contains the number of parameters that are passed to the external user-defined function, including `argv[0]`.

The following figure shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result.

```
#include <stdlib.h>
#include <stdio.h>

main(argc,argv)
  int argc;
  char *argv[];
{
  /******
  /* Assume that the user-defined function invocation*/
  /* included 2 input parameters in the parameter */
  /* list. Also assume that the definition includes */
  /* the SCRATCHPAD, FINAL CALL, and DBINFO options, */
  /* so DB2 passes the scratchpad, calltype, and */
  /* dbinfo parameters. */
  /* The argv vector contains these entries: */
  /*   argv[0]           1   load module name */
  /*   argv[1-2]        2   input parms */
  /*   argv[3]          1   result parm */
  /*   argv[4-5]        2   null indicators */
  /*   argv[6]          1   result null indicator */
  /*   argv[7]          1   SQLSTATE variable */
  /*   argv[8]          1   qualified func name */
  /*   argv[9]          1   specific func name */
  /*   argv[10]         1   diagnostic string */
  /*   argv[11]         1   scratchpad */
  /*   argv[12]         1   call type */
  /*   argv[13]         + 1 dbinfo */
  /*           ----- */
  /*           14   for the argc variable */
  /******
  if argc<>14
  {
  :
  :
  /******
  /* This section would contain the code executed if the */
  /* user-defined function is invoked with the wrong number */
  /* of parameters. */
  /******
  }

  /******
  /* Assume the first parameter is an integer. */
  /* The following code shows how to copy the integer*/
  /* parameter into the application storage. */
  /******
```

```

int parm1;
parm1 = *(int *) argv[1];

/*****
/* Access the null indicator for the first
/* parameter on the invoked user-defined function
/* as follows:
*****/
short int ind1;
ind1 = *(short int *) argv[4];

/*****
/* Use the following expression to assign
/* 'xxxxx' to the SQLSTATE returned to caller on
/* the SQL statement that contains the invoked
/* user-defined function.
*****/
strcpy(argv[7],"xxxxx");

/*****
/* Obtain the value of the qualified function
/* name with this expression.
*****/
char f_func[28];
strcpy(f_func,argv[8]);

/*****
/* Obtain the value of the specific function
/* name with this expression.
*****/
char f_spec[19];
strcpy(f_spec,argv[9]);

/*****
/* Use the following expression to assign
/* 'yyyyyyy' to the diagnostic string returned
/* in the SQLCA associated with the invoked
/* user-defined function.
*****/
strcpy(argv[10],"yyyyyyy");

/*****
/* Use the following expression to assign the
/* result of the function.
*****/
char l_result[11];
strcpy(argv[3],l_result);

:
}

```

The following figure shows the parameter conventions for a user-defined scalar function written as a C subprogram that receives two parameters and returns one result.

```

#pragma runopts(plist(os))
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sqludf.h>

void myfunc(long *parm1, char parm2[11], char result[11],
            short *f_ind1, short *f_ind2, short *f_indr,
            char udf_sqlstate[6], char udf_fname[138],
            char udf_specname[129], char udf_msgtext[71],
            struct sqludf_scratchpad *udf_scratchpad,
            long *udf_call_type,

```



```

        struct sql_dbinfo *udf_dbinfo);
{
    /******
    /* Declare local copies of parameters          */
    /******
    int l_p1;
    char l_p2[11];
    short int l_ind1;
    short int l_ind2;
    char ludf_sqlstate[6];      /* SQLSTATE          */
    char ludf_fname[138];      /* function name    */
    char ludf_specname[129];    /* specific function name */
    char ludf_msgtext[71]      /* diagnostic message text*/
    sqludf_scratchpad *ludf_scratchpad; /* scratchpad      */
    long *ludf_call_type;      /* call type        */
    sqludf_dbinfo *ludf_dbinfo /* dbinfo           */
    /******
    /* Copy each of the parameters in the parameter */
    /* list into a local variable to demonstrate   */
    /* how the parameters can be referenced.        */
    /******

    l_p1 = *parm1;
    strcpy(l_p2,parm2);
    l_ind1 = *f_ind1;
    l_ind1 = *f_ind2;
    strcpy(ludf_sqlstate,udf_sqlstate);
    strcpy(ludf_fname,udf_fname);
    strcpy(ludf_specname,udf_specname);
    l_udf_call_type = *udf_call_type;
    strcpy(ludf_msgtext,udf_msgtext);
    memcpy(&ludf_scratchpad,udf_scratchpad,sizeof(ludf_scratchpad));
    memcpy(&ludf_dbinfo,udf_dbinfo,sizeof(ludf_dbinfo));

:
}

```

The following figure shows the parameter conventions for a user-defined scalar function that is written as a C++ subprogram that receives two parameters and returns one result. This example demonstrates that you must use an extern "C" modifier to indicate that you want the C++ subprogram to receive parameters according to the C linkage convention. This modifier is necessary because the CEEPIPI CALL_SUB interface, which DB2 uses to call the user-defined function, passes parameters using the C linkage convention.

```

#pragma runopts(plist(os))
#include <stdlib.h>
#include <stdio.h>
#include <sqludf.h>

extern "C" void myfunc(long *parm1, char parm2[11],
    char result[11], short *f_ind1, short *f_ind2, short *f_indr,
    char udf_sqlstate[6], char udf_fname[138],
    char udf_specname[129], char udf_msgtext[71],
    struct sqludf_scratchpad *udf_scratchpad,
    long *udf_call_type,
    struct sql_dbinfo *udf_dbinfo);

{
    /******
    /* Define local copies of parameters.          */
    /******
    int l_p1;
    char l_p2[11];
    short int l_ind1;
    short int l_ind2;

```

```

char ludf_sqlstate[6];      /* SQLSTATE          */
char ludf_fname[138];     /* function name     */
char ludf_specname[129]; /* specific function name */
char ludf_msgtext[71]    /* diagnostic message text*/
sqludf_scratchpad *ludf_scratchpad; /* scratchpad      */
long *ludf_call_type;     /* call type        */
sqludf_dbinfo *ludf_dbinfo /* dbinfo          */
/*****/
/* Copy each of the parameters in the parameter */
/* list into a local variable to demonstrate */
/* how the parameters can be referenced.      */
/*****/
l_p1 = *parm1;
strcpy(l_p2,parm2);
l_ind1 = *f_ind1;
l_ind1 = *f_ind2;
strcpy(ludf_sqlstate,udf_sqlstate);
strcpy(ludf_fname,udf_fname);
strcpy(ludf_specname,udf_specname);
l_udf_call_type = *udf_call_type;
strcpy(ludf_msgtext,udf_msgtext);
memcpy(&ludf_scratchpad,udf_scratchpad,sizeof(ludf_scratchpad));
memcpy(&ludf_dbinfo,udf_dbinfo,sizeof(ludf_dbinfo));

:
}

```

COBOL: The following figure shows the parameter conventions for a user-defined table function that is written as a main program that receives two parameters and returns two results. For a COBOL user-defined function that is a subprogram, the conventions are the same.

```

CBL APOST,RES,RENT
  IDENTIFICATION DIVISION.

:

:
  DATA DIVISION.

:

:
  LINKAGE SECTION.
  *****
  * Declare each of the parameters          *
  *****
  01  UDFPARAM1 PIC S9(9) USAGE COMP.
  01  UDFPARAM2 PIC X(10).

:

:
  *****
  * Declare these variables for result parameters *
  *****
  01  UDFRESULT1 PIC X(10).
  01  UDFRESULT2 PIC X(10).

:

:
  *****
  * Declare a null indicator for each parameter *
  *****
  01  UDF-IND1 PIC S9(4) USAGE COMP.
  01  UDF-IND2 PIC S9(4) USAGE COMP.

:

:
  *****
  * Declare a null indicator for result parameter *
  *****
  01  UDF-RIND1 PIC S9(4) USAGE COMP.
  01  UDF-RIND2 PIC S9(4) USAGE COMP.

```

⋮

```
*****
* Declare the SQLSTATE that can be set by the          *
* user-defined function                               *
*****
01 UDF-SQLSTATE PIC X(5).
*****
* Declare the qualified function name                 *
*****
01 UDF-FUNC.
   49 UDF-FUNC-LEN PIC 9(4) USAGE BINARY.
   49 UDF-FUNC-TEXT PIC X(137).
*****
* Declare the specific function name                 *
*****
01 UDF-SPEC.
   49 UDF-SPEC-LEN PIC 9(4) USAGE BINARY.
   49 UDF-SPEC-TEXT PIC X(128).
*****
* Declare SQL diagnostic message token               *
*****
01 UDF-DIAG.
   49 UDF-DIAG-LEN PIC 9(4) USAGE BINARY.
   49 UDF-DIAG-TEXT PIC X(1000).
*****
* Declare the scratchpad                             *
*****
01 UDF-SCRATCHPAD.
   49 UDF-SPAD-LEN PIC 9(9) USAGE BINARY.
   49 UDF-SPAD-TEXT PIC X(100).
*****
* Declare the call type                              *
*****
01 UDF-CALL-TYPE PIC 9(9) USAGE BINARY.
*****
* CONSTANTS FOR DB2-EBCODING-SCHEME.                 *
*****
77 SQLUDF-ASCII PIC 9(9) VALUE 1.
77 SQLUDF-EBCDIC PIC 9(9) VALUE 2.
77 SQLUDF-UNICODE PIC 9(9) VALUE 3.
*****
* Structure used for DBINFO                           *
*****
01 SQLUDF-DBINFO.
*   location name length
   05 DBNAMELEN PIC 9(4) USAGE BINARY.
*   location name
   05 DBNAME PIC X(128).
*   authorization ID length
   05 AUTHIDLEN PIC 9(4) USAGE BINARY.
*   authorization ID
   05 AUTHID PIC X(128).
*   environment CCSID information
   05 CODEPG PIC X(48).
   05 CDPG-DB2 REDEFINES CODEPG.
      10 DB2-CCSIDS OCCURS 3 TIMES.
         15 DB2-SBCS   PIC 9(9) USAGE BINARY.
         15 DB2-DBCS   PIC 9(9) USAGE BINARY.
         15 DB2-MIXED  PIC 9(9) USAGE BINARY.
      10 ENCODING-SCHEME PIC 9(9) USAGE BINARY.
      10 RESERVED     PIC X(8).
* other platform-specific deprecated CCSID structures not included here
*   schema name length
   05 TBSCHMALEN PIC 9(4) USAGE BINARY.
*   schema name
   05 TBSCHMA PIC X(128).
```

```

*      table name length
05  TBNAMELEN PIC 9(4) USAGE BINARY.
*      table name
05  TBNAME PIC X(128).
*      column name length
05  COLNAMELEN PIC 9(4) USAGE BINARY.
*      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) USAGE BINARY.
*      number of entries in tfcolumn list array (tfcolumn, below)
05  NUMTFCOL PIC 9(4) USAGE BINARY.
*      reserved for expansion
05  RESD1 PIC X(26).
*      tfcolumn will be allocated dynamically if TF is defined
*      otherwise this will be a null pointer
05  TFCOLUMN USAGE IS POINTER.
*      Application identifier
05  APPL-ID USAGE IS POINTER.
*      reserved for expansion
05  RESD2 PIC X(20).      *
PROCEDURE DIVISION USING UDFPARAM1, UDFPARAM2, UDFRESULT1,
                        UDFRESULT2, UDF-IND1, UDF-IND2,
                        UDF-RIND1, UDF-RIND2,
                        UDF-SQLSTATE, UDF-FUNC, UDF-SPEC,
                        UDF-DIAG, UDF-SCRATCHPAD,
                        UDF-CALL-TYPE, SQLUDF-DBINFO.

```

PL/I: The following figure shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result. For a PL/I user-defined function that is a subprogram, the conventions are the same.

```

*PROCESS SYSTEM(MVS);
MYMAIN: PROC(UDF_PARM1, UDF_PARM2, UDF_RESULT,
            UDF_IND1, UDF_IND2, UDF_INDR,
            UDF_SQLSTATE, UDF_NAME, UDF_SPEC_NAME,
            UDF_DIAG_MSG, UDF_SCRATCHPAD,
            UDF_CALL_TYPE, UDF_DBINFO)
OPTIONS(MAIN NOEXECOPS REENTRANT);

DCL UDF_PARM1 BIN FIXED(31);      /* first parameter      */
DCL UDF_PARM2 CHAR(10);          /* second parameter     */
DCL UDF_RESULT CHAR(10);        /* result parameter     */
DCL UDF_IND1 BIN FIXED(15);     /* indicator for 1st parm */
DCL UDF_IND2 BIN FIXED(15);     /* indicator for 2nd parm */
DCL UDF_INDR BIN FIXED(15);     /* indicator for result  */
DCL UDF_SQLSTATE CHAR(5);      /* SQLSTATE returned to DB2 */
DCL UDF_NAME CHAR(137) VARYING; /* Qualified function name */
DCL UDF_SPEC_NAME CHAR(128) VARYING; /* Specific function name */
DCL UDF_DIAG_MSG CHAR(70) VARYING; /* Diagnostic string */
DCL 01 UDF_SCRATCHPAD          /* Scratchpad */
    03 UDF_SPAD_LEN BIN FIXED(31),
    03 UDF_SPAD_TEXT CHAR(100);
DCL UDF_CALL_TYPE BIN FIXED(31); /* Call Type */
DCL DBINFO PTR;
/* CONSTANTS FOR DB2_ENCODING_SCHEME */
DCL SQLUDF_ASCII BIN FIXED(15) INIT(1);
DCL SQLUDF_EBCDIC BIN FIXED(15) INIT(2);
DCL SQLUDF_MIXED BIN FIXED(15) INIT(3);


```

```

DCL 01 UDF_DBINFO BASED(DBINFO),          /* Dbinfo          */
      03 UDF_DBINFO_LLEN BIN FIXED(15),    /* location length */
      03 UDF_DBINFO_LOC CHAR(128),        /* location name   */
      03 UDF_DBINFO_ALEN BIN FIXED(15),    /* auth ID length  */
      03 UDF_DBINFO_AUTH CHAR(128),       /* authorization ID */
      03 UDF_DBINFO_CDPG,                  /* environment CCSID info */
      05 DB2_CCSDS(3),
      07 R1 BIN FIXED(15), /* Reserved          */
      07 DB2_SBCS BIN FIXED(15), /* SBCS CCSID      */
      07 R2 BIN FIXED(15), /* Reserved          */
      07 DB2_DBCS BIN FIXED(15), /* DBCS CCSID      */
      07 R3 BIN FIXED(15), /* Reserved          */
      07 DB2_MIXED BIN FIXED(15), /* MIXED CCSID     */
      05 DB2_ENCODING_SCHEME BIN FIXED(31),
      05 DB2_CCSDS_RESERVED CHAR(8),
      03 UDF_DBINFO_SLEN BIN FIXED(15),    /* schema length   */
      03 UDF_DBINFO_SCHEMA CHAR(128),     /* schema name     */
      03 UDF_DBINFO_TLEN BIN FIXED(15),    /* table length    */
      03 UDF_DBINFO_TABLE CHAR(128),      /* table name      */
      03 UDF_DBINFO_CLEN BIN FIXED(15),    /* column length   */
      03 UDF_DBINFO_COLUMN CHAR(128),     /* column name     */
      03 UDF_DBINFO_RELVER CHAR(8),       /* DB2 release level */
      03 UDF_DBINFO_RESERVED CHAR(2),     /* reserved        */
      03 UDF_DBINFO_PLATFORM BIN FIXED(31), /* database platform */
      03 UDF_DBINFO_NUMTFCOL BIN FIXED(15), /* # of TF columns used */
      03 UDF_DBINFO_RESERV1 CHAR(26),     /* reserved        */
      03 UDF_DBINFO_TFCOLUMN PTR,         /* -> TFcolumn list */
      03 UDF_DBINFO_APPLID PTR,          /* -> application id */
      03 UDF_DBINFO_RESERV2 CHAR(20);    /* reserved        */
:

```

Related reference:

 CREATE FUNCTION (external scalar) (DB2 SQL)

Making a user-defined function reentrant

A reentrant user-defined function is a function for which a single copy of the function can be used concurrently by two or more processes.

Compiling and link-editing your user-defined function as reentrant is recommended. (For an assembler program, you must also code the user-defined function to be reentrant.) Reentrant user-defined functions have the following advantages:

- The operating system does not need to load the user-defined function into storage every time the user-defined function is called.
- Multiple tasks in a WLM-established stored procedures address space can share a single copy of the user-defined function. This decreases the amount of virtual storage that is needed for code in the address space.

Preparing user-defined functions that contain multiple programs: If your user-defined function consists of several programs, you must bind each program that contains SQL statements into a separate package. The definer of the user-defined function must have EXECUTE authority for all packages that are part of the user-defined function.

When the primary program of a user-defined function calls another program, DB2 uses the CURRENT PACKAGE PATH special register to determine the list of collections to search for the called program's package. The primary program can change this collection ID by executing the statement SET CURRENT PACKAGE PATH.

If the value of CURRENT PACKAGE PATH is blank or an empty string, DB2 uses the CURRENT PACKAGESET special register to determine the collection to search for the called program's package. The primary program can change this value by executing the statement SET CURRENT PACKAGESET.

If both special registers CURRENT PACKAGE PATH and CURRENT PACKAGESET contain a blank value, DB2 uses the method described in "Binding an application plan" on page 949 to search for the package.

Special registers in a user-defined function or a stored procedure

You can use all special registers in a user-defined function or a stored procedure. However, you can modify only some of those special registers.

After a user-defined function or a stored procedure completes, DB2 restores all special registers to the values they had before invocation.

The following table shows information that you need when you use special registers in a user-defined function or stored procedure.

Table 91. Characteristics of special registers in a user-defined function or a stored procedure

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT APPLICATION ENCODING SCHEME	The value of bind option ENCODING for the user-defined function or stored procedure package	The value of bind option ENCODING for the user-defined function or stored procedure package	Yes
CURRENT CLIENT_ACCTNG	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT CLIENT_APPLNAME	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT CLIENT_USERID	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT CLIENT_WRKSTNNAME	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
CURRENT DATE	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT DEBUG MODE	Inherited from the invoking application	DISALLOW	Yes
CURRENT DECFLOAT ROUNDING MODE	Inherited from the invoking application	The value of bind option ROUNDING for the user-defined function or stored procedure package	Yes
CURRENT DEGREE	CURRENT DEGREE ²	The value of field CURRENT DEGREE on installation panel DSNTIP8	Yes
CURRENT EXPLAIN MODE	Inherited from the invoking application	NO	Yes

Table 91. Characteristics of special registers in a user-defined function or a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT GET_ACCEL_ARCHIVE	Inherited from the invoking application	System default value	Yes
CURRENT LOCALE LC_CTYPE	Inherited from the invoking application	The value of field CURRENT LC_CTYPE on installation panel DSNTIPF	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Inherited from the invoking application	System default value	Yes
CURRENT MEMBER	New value for each SET <i>host-variable</i> =CURRENT MEMBER statement	New value for each SET <i>host-variable</i> =CURRENT MEMBER statement	Not applicable ⁵
CURRENT OPTIMIZATION HINT	The value of bind option OPTHINT for the user-defined function or stored procedure package or inherited from the invoking application ⁶	The value of bind option OPTHINT for the user-defined function or stored procedure package	Yes
CURRENT PACKAGE PATH	An empty string if the routine was defined with a COLLID value; otherwise, inherited from the invoking application ⁴	An empty string, regardless of whether a COLLID value was specified for the routine ⁴	Yes
CURRENT PACKAGESET	Inherited from the invoking application ³	Inherited from the invoking application ³	Yes
CURRENT PATH	The value of bind option PATH for the user-defined function or stored procedure package or inherited from the invoking application ⁶	The value of bind option PATH for the user-defined function or stored procedure package	Yes
CURRENT PRECISION	Inherited from the invoking application	The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4	Yes
CURRENT QUERY ACCELERATION	Inherited from the invoking application	System default value	Yes
CURRENT REFRESH AGE	Inherited from the invoking application	System default value	Yes
CURRENT ROUTINE VERSION	Inherited from the invoking application	The empty string	Yes
CURRENT RULES	Inherited from the invoking application	The value of bind option SQLRULES for the plan that invokes a user-defined function or stored procedure	Yes
CURRENT SCHEMA	Inherited from the invoking application	The value of CURRENT SCHEMA when the routine is entered	Yes
CURRENT SERVER	Inherited from the invoking application	Inherited from the invoking application	Yes

Table 91. Characteristics of special registers in a user-defined function or a stored procedure (continued)

Special register	Initial value when INHERIT SPECIAL REGISTERS option is specified	Initial value when DEFAULT SPECIAL REGISTERS option is specified	Routine can use SET statement to modify?
CURRENT SQLID	The primary authorization ID of the application process or inherited from the invoking application ⁷	The primary authorization ID of the application process	Yes ⁸
CURRENT TIME	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT TIMESTAMP	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT TIMESTAMP WITH TIME ZONE	New value for each SQL statement in the user-defined function or stored procedure package ¹	New value for each SQL statement in the user-defined function or stored procedure package ¹	Not applicable ⁵
CURRENT TIME ZONE	Inherited from the invoking application	Inherited from the invoking application	Not applicable ⁵
ENCRYPTION PASSWORD	Inherited from the invoking application	Inherited from the invoking application	Yes
SESSION TIME ZONE	Inherited from the invoking application	The value of CURRENT TIME ZONE when the routine is entered	Yes
SESSION_USER or USER	Primary authorization ID of the application process	Primary authorization ID of the application process	Not applicable ⁵

Notes:

1. If the user-defined function or stored procedure is invoked within the scope of a trigger, DB2 uses the timestamp for the triggering SQL statement as the timestamp for all SQL statements in the package.
2. DB2 allows parallelism at only one level of a nested SQL statement. If you set the value of the CURRENT DEGREE special register to ANY, and parallelism is disabled, DB2 ignores the CURRENT DEGREE value.
3. If the routine definition includes a specification for COLLID, DB2 sets CURRENT PACKAGESET to the value of COLLID. If both CURRENT PACKAGE PATH and COLLID are specified, the CURRENT PACKAGE PATH value takes precedence and COLLID is ignored.
4. If the function definition includes a specification for PACKAGE PATH, DB2 sets CURRENT PACKAGE PATH to the value of PACKAGE PATH.
5. Not applicable because no SET statement exists for the special register.
6. If a program within the scope of the invoking program issues a SET statement for the special register before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, the special register contains the value that is set by the bind option for the user-defined function or stored procedure package.
7. If a program within the scope of the invoking program issues a SET CURRENT SQLID statement before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, CURRENT SQLID contains the authorization ID of the application process.
8. If the user-defined function or stored procedure package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed. However, it does not affect the authorization ID that is used for the dynamic SQL statements in the package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements.

Related concepts:

➤ DYNAMICRULES bind option (DB2 Application programming and SQL)

Related reference:

➤ BIND and REBIND options for packages and plans (DB2 Commands)

➤ Special registers (DB2 SQL)

Accessing transition tables in a user-defined function or stored procedure

If you want to refer to the entire set of rows that a triggering SQL statement modifies, rather than to individual rows, use a transition table. You can reference a transition table in user-defined functions and procedures that are invoked from a trigger.

This topic describes how to access transition variables in a user-defined function, but the same techniques apply to a stored procedure.

To access transition tables in a user-defined function, use table locators, which are pointers to the transition tables. You declare table locators as input parameters in the CREATE FUNCTION statement using the TABLE LIKE *table-name* AS LOCATOR clause.

To access transition tables in a user-defined function or stored procedure:

1. Declare input parameters to receive table locators. You must define each parameter that receives a table locator as an unsigned 4-byte integer.
2. Declare table locators. You can declare table locators in assembler, C, C++, COBOL, PL/I, and in an SQL procedure compound statement.
3. Declare a cursor to access the rows in each transition table.
4. Assign the input parameter values to the table locators.
5. Access rows from the transition tables using the cursors that are declared for the transition tables.

The following examples show how a user-defined function that is written in C, C++, COBOL, or PL/I accesses a transition table for a trigger. The transition table, NEWEMP, contains modified rows of the employee sample table. The trigger is defined like this:

```
CREATE TRIGGER EMPRAISE
  AFTER UPDATE ON EMP
  REFERENCING NEW TABLE AS NEWEMPS
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    VALUES (CHECKEMP(TABLE NEWEMPS));
  END;
```

The user-defined function definition looks like this:

```
CREATE FUNCTION CHECKEMP(TABLE LIKE EMP AS LOCATOR)
  RETURNS INTEGER
  EXTERNAL NAME 'CHECKEMP'
  PARAMETER STYLE SQL
  LANGUAGE language;
```

Assembler: The following example shows how an assembler program accesses rows of transition table NEWEMPS.

```
CHECKEMP CSECT
          SAVE   (14,12)          ANY SAVE SEQUENCE
          LR     R12,R15          CODE ADDRESSABILITY
```

```

        USING CHECKEMP,R12      TELL THE ASSEMBLER
        LR   R7,R1             SAVE THE PARM POINTER
        USING PARMAREA,R7      SET ADDRESSABILITY FOR PARMS
        USING SQLDSECT,R8      ESTABLISH ADDRESSIBILITY TO SQLDSECT
        L    R6,PROGSIze       GET SPACE FOR USER PROGRAM
        GETMAIN R,LV=(6)       GET STORAGE FOR PROGRAM VARIABLES
        LR   R10,R1            POINT TO THE ACQUIRED STORAGE
        LR   R2,R10            POINT TO THE FIELD
        LR   R3,R6             GET ITS LENGTH
        SR   R4,R4             CLEAR THE INPUT ADDRESS
        SR   R5,R5             CLEAR THE INPUT LENGTH
        MVCL R2,R4             CLEAR OUT THE FIELD
        ST   R13,FOUR(R10)     CHAIN THE SAVEAREA PTRS
        ST   R10,EIGHT(R13)    CHAIN SAVEAREA FORWARD
        LR   R13,R10           POINT TO THE SAVEAREA
        USING PROGAREA,R13     SET ADDRESSABILITY
        ST   R6,GETLENTH      SAVE THE LENGTH OF THE GETMAIN

:
*****
* Declare table locator host variable TRIGTBL *
*****
TRIGTBL SQL TYPE IS TABLE LIKE EMP AS LOCATOR
*****
* Declare a cursor to retrieve rows from the transition *
* table *
*****
        EXEC SQL DECLARE C1 CURSOR FOR
                SELECT LASTNAME FROM TABLE(:TRIGTBL LIKE EMP)
                WHERE SALARY > 100000
*****
* Copy table locator for trigger transition table *
*****
        L    R2,TABLOC         GET ADDRESS OF LOCATOR
        L    R2,0(0,R2)        GET LOCATOR VALUE
        ST   R2,TRIGTBL
        EXEC SQL OPEN C1
        EXEC SQL FETCH C1 INTO :NAME

:
:
        EXEC SQL CLOSE C1
:
:
PROGAREA DSECT
SAVEAREA DS 18F
GETLENTH DS A
:
NAME DS CL24
:
:
        DS 0D
PROGSIze EQU *-PROGAREA
PARMAREA DSECT
TABLOC DS A
:
:
        END CHECKEMP

```

X
X

C or C++: The following example shows how a C or C++ program accesses rows of transition table NEWEMPS.

```

int CHECK_EMP(int trig_tbl_id)
{
:
:
:
/*****
/* Declare table locator host variable trig_tbl_id */
/*****
EXEC SQL BEGIN DECLARE SECTION;

```

```

        SQL TYPE IS TABLE LIKE EMP AS LOCATOR trig_tbl_id;
        char name[25];
EXEC SQL END DECLARE SECTION;

:
:
/*****
/* Declare a cursor to retrieve rows from the transition */
/* table */
/*****
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT NAME FROM TABLE(:trig_tbl_id LIKE EMPLOYEE)
        WHERE SALARY > 100000;
/*****
/* Fetch a row from transition table */
/*****
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 INTO :name;

:
EXEC SQL CLOSE C1;

:
:
}

```

COBOL: The following example shows how a COBOL program accesses rows of transition table NEWEMPS.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHECKEMP.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NAME PIC X(24).

:

LINKAGE SECTION.
*****
* Declare table locator host variable TRIG-TBL-ID *
*****
01 TRIG-TBL-ID SQL TYPE IS TABLE LIKE EMP AS LOCATOR.

:

PROCEDURE DIVISION USING TRIG-TBL-ID.

:

*****
* Declare cursor to retrieve rows from transition table *
*****
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT NAME FROM TABLE(:TRIG-TBL-ID LIKE EMP)
        WHERE SALARY > 100000 END-EXEC.
*****
* Fetch a row from transition table *
*****
EXEC SQL OPEN C1 END-EXEC.
EXEC SQL FETCH C1 INTO :NAME END-EXEC.

:

EXEC SQL CLOSE C1 END-EXEC.

:

PROG-END.
GOBACK.

```

PL/I: The following example shows how a PL/I program accesses rows of transition table NEWEMPS.

```
CHECK_EMP: PROC(TRIG_TBL_ID) RETURNS(BIN FIXED(31))
            OPTIONS(MAIN NOEXECOPS REENTRANT);
/*****
/* Declare table locator host variable TRIG_TBL_ID */
*****/
DECLARE TRIG_TBL_ID SQL TYPE IS TABLE LIKE EMP AS LOCATOR;
DECLARE NAME CHAR(24);

:
/*****
/* Declare a cursor to retrieve rows from the      */
/* transition table                               */
*****/
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT NAME FROM TABLE(:TRIG_TBL_ID LIKE EMP)
        WHERE SALARY > 100000;
/*****
/* Retrieve rows from the transition table        */
*****/
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 INTO :NAME;

:
EXEC SQL CLOSE C1;

:
END CHECK_EMP;
```

Preparing an external user-defined function for execution

Because an external user-defined function is written in a programming language, preparing it is similar to the way that you prepare any other application program.

To prepare an external user-defined function for execution:

1. Precompile the user-defined function program and bind the DBRM into a package. You need to do this only if your user-defined function contains SQL statements. You do not need to bind a plan for the user-defined function.
2. Compile the user-defined function program and link-edit it with Language Environment and RRSAP.

You must compile the program with a compiler that supports Language Environment and link-edit the appropriate Language Environment components with the user-defined function. You must also link-edit the user-defined function with RRSAP.

The program preparation JCL samples DSNHASM, DSNHC, DSNHCPP, DSNHICOB, and DSNHPLI show you how to precompile, compile, and link-edit assembler, C, C++, COBOL, and PL/I DB2 programs. For object-oriented programs in C++, see JCL sample DSNHCPP2 for program preparation hints.

3. For a user-defined function that contains SQL statements, grant EXECUTE authority on the user-defined function package to the function definer.

Abnormal termination of an external user-defined function

If an external user-defined function abnormally terminates, your program receives SQLCODE -430 for invoking the statement.

DB2 also performs the following actions:

- Places the unit of work that contains the invoking statement in a must-rollback state.
- Stops the user-defined function, and subsequent calls fail, in either of the following situations:
 - The number of abnormal terminations equals the STOP AFTER *n* FAILURES value for the user-defined function.
 - If the STOP AFTER *n* FAILURES option is not specified, the number of abnormal terminations equals the default MAX ABEND COUNT value for the subsystem.

You should include code in your program to check for a user-defined function abend and to roll back the unit of work that contains the user-defined function invocation.

Saving information between invocations of a user-defined function by using a scratchpad

If you create a scratchpad for a reentrant user-defined function, DB2 can use it to preserve information between invocations of the function.

You can use a scratchpad to save information between invocations of a user-defined function. To indicate that a scratchpad should be allocated when the user-defined function executes, the function definer specifies the SCRATCHPAD parameter in the CREATE FUNCTION statement.

The scratchpad consists of a 4-byte length field, followed by the scratchpad area. The definer can specify the length of the scratchpad area in the CREATE FUNCTION statement. The specified length does not include the length field. The default size is 100 bytes. DB2 initializes the scratchpad for each function to binary zeros at the beginning of execution for each subquery of an SQL statement and does not examine or change the content thereafter. On each invocation of the user-defined function, DB2 passes the scratchpad to the user-defined function. You can therefore use the scratchpad to preserve information between invocations of a reentrant user-defined function.

The following example demonstrates how to enter information in a scratchpad for a user-defined function defined like this:

```
CREATE FUNCTION COUNTER()
  RETURNS INT
  SCRATCHPAD
  FENCED
  NOT DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
  LANGUAGE C
  PARAMETER STYLE SQL
  EXTERNAL NAME 'UDFCTR';
```

The scratchpad length is not specified, so the scratchpad has the default length of 100 bytes, plus 4 bytes for the length field. The user-defined function increments an integer value and stores it in the scratchpad on each execution.

```
#pragma linkage(ctr,fetchable)
#include <stdlib.h>
#include <stdio.h>
/* Structure scr defines the passed scratchpad for function ctr */
struct scr {
  long len;
  long countr;
```

```

        char not_used[96];
    };
/*****
/* Function ctr: Increments a counter and reports the value */
/*                from the scratchpad.                      */
/*                */
/*    Input: None                                          */
/*    Output: INTEGER out      the value from the scratchpad */
*****/
void ctr(
    long *out,                /* Output answer (counter) */
    short *outnull,          /* Output null indicator   */
    char *sqlstate,         /* SQLSTATE                */
    char *funcname,         /* Function name           */
    char *specname,         /* Specific function name  */
    char *mesgtext,         /* Message text insert     */
    struct scr *scratchptr) /* Scratchpad              */
{
    *out = ++scratchptr->ctr; /* Increment counter and   */
                          /* copy to output variable */
    *outnull = 0;          /* Set output null indicator*/
    return;
}
/* end of user-defined function ctr */

```

Example of creating and using a user-defined scalar function

You can create a user-defined scalar function that gets input from a table and puts the output in a table.

Suppose that your organization needs a user-defined scalar function that calculates the bonus that each employee receives. All employee data, including salaries, commissions, and bonuses, is kept in the employee table, EMP. The input fields for the bonus calculation function are the values of the SALARY and COMM columns. The output from the function goes into the BONUS column. Because this function gets its input from a DB2 table and puts the output in a DB2 table, a convenient way to manipulate the data is through a user-defined function.

The user-defined function's definer and invoker determine that this new user-defined function should have these characteristics:

- The user-defined function name is CALC_BONUS.
- The two input fields are of type DECIMAL(9,2).
- The output field is of type DECIMAL(9,2).
- The program for the user-defined function is written in COBOL and has a load module name of CBONUS.

Because no built-in function or user-defined function exists on which to build a sourced user-defined function, the function implementer must code an external user-defined function. The implementer performs the following steps:

- Writes the user-defined function, which is a COBOL program
- Precompiles, compiles, and links the program
- Binds a package if the user-defined function contains SQL statements
- Tests the program thoroughly
- Grants execute authority on the user-defined function package to the definer

The user-defined function definer executes this CREATE FUNCTION statement to register CALC_BONUS to DB2:

```
CREATE FUNCTION CALC_BONUS(DECIMAL(9,2),DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  EXTERNAL NAME 'CBONUS'
  PARAMETER STYLE SQL
  LANGUAGE COBOL;
```

The definer then grants execute authority on CALC_BONUS to all invokers.

User-defined function invokers write and prepare application programs that invoke CALC_BONUS. An invoker might write a statement like this, which uses the user-defined function to update the BONUS field in the employee table:

```
UPDATE EMP
  SET BONUS = CALC_BONUS(SALARY,COMM);
```

An invoker can execute this statement either statically or dynamically.

User-defined function samples that ship with DB2

To assist you in defining, implementing, and invoking your user-defined functions, DB2 provides a number of sample user-defined functions. All sample user-defined function code is in data set DSN10.SDSNSAMP.

The following table summarizes the characteristics of the sample user-defined functions.

Table 92. User-defined function samples shipped with DB2

User-defined function name	Language	Member that contains source code	Purpose
ALTDATE ¹	C	DSN8DUAD	Converts the current date to a user-specified format
ALTDATE ²	C	DSN8DUCD	Converts a date from one format to another
ALTTIME ³	C	DSN8DUAT	Converts the current time to a user-specified format
ALTTIME ⁴	C	DSN8DUCT	Converts a time from one format to another
DAYNAME	C++	DSN8EUDN	Returns the day of the week for a user-specified date
MONTHNAME	C++	DSN8EUMN	Returns the month for a user-specified date
CURRENCY	C	DSN8DUCY	Formats a floating-point number as a currency value
TABLE_NAME	C	DSN8DUTI	Returns the unqualified table name for a table, view, or alias
TABLE_QUALIF	C	DSN8DUTI	Returns the qualifier for a table, view, or alias
TABLE_LOCATION	C	DSN8DUTI	Returns the location for a table, view, or alias
WEATHER	C	DSN8DUWF	Returns a table of weather information from a EBCDIC data set

Notes:

1. This version of ALTDATE has one input parameter, of type VARCHAR(13).
2. This version of ALTDATE has three input parameters, of type VARCHAR(17), VARCHAR(13), and VARCHAR(13).
3. This version of ALTTIME has one input parameter, of type VARCHAR(14).
4. This version of ALTTIME has three input parameters, of type VARCHAR(11), VARCHAR(14), and VARCHAR(14).

Member DSN8DUWC contains a client program that shows you how to invoke the WEATHER user-defined table function.

Member DSNTEJ2U shows you how to define and prepare the sample user-defined functions and the client program.

Related concepts:

[➤ Job DSNTEJ2U \(DB2 Installation and Migration\)](#)

[➤ Sample user-defined functions \(DB2 SQL\)](#)

Creating a stored procedure

A *stored procedure* is executable code that can be called by other programs. The process for creating one depends on the type of procedure.

Before you can use the following types of stored procedures, you must configure your DB2 for z/OS subsystem for running stored procedures during installation or configure your DB2 for z/OS subsystem for running stored procedures during migration.

- External stored procedures
- External SQL procedures
- Native SQL procedures that satisfy at least one of the following conditions:
 - The native SQL procedure calls at least one external stored procedure, external SQL procedure, or user-defined function.
 - The native SQL procedure is defined with ALLOW DEBUG MODE or DISALLOW DEBUG MODE.
- DB2-supplied stored procedures

You can create one of the following types of stored procedures:

External stored procedures

A procedure that is written in a host language.

External SQL procedures

A procedure whose body is written entirely in SQL, but is created, implemented, and executed like other external stored procedures.

Native SQL procedures

A procedure with a procedural body that is written entirely in SQL and is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not have an associated external application program.

To create a stored procedure, perform one of the following actions:

- “Creating a native SQL procedure” on page 572
- “Creating an external SQL procedure” on page 600
- “Creating an external stored procedure” on page 615

Related concepts:

“External stored procedures” on page 569

“SQL procedures” on page 565

Related tasks:

[➤ Implementing DB2 stored procedures \(DB2 Administration Guide\)](#)

Related reference:

[➤ DB2 for z/OS Exchange](#)

Stored procedures

A *stored procedure* is a compiled program that can execute SQL statements and is stored at a local or remote DB2 server. You can invoke a stored procedure from an application program or from the command line processor. A single call to a stored procedure from a client application can access the database at the server several times.

A typical stored procedure contains two or more SQL statements and some manipulative or logical processing in a host language or SQL procedure statements. You can call stored procedures from other applications or from the command line. DB2 provides some stored procedures, but you can also create your own.

A stored procedure provides a common piece of code that is written only once and is maintained in a single instance that can be called from several different applications. Host languages can easily call procedures that exist on a local system, and SQL can call stored procedures that exist on remote systems. In fact, a major benefit of procedures in SQL is that they can be used to enhance the performance characteristics of distributed applications. With stored procedures, you can avoid network transfer of large amounts of data obtained as part of intermediate results in a long sequence of queries.

The following diagram illustrates the processing for an application that does not use stored procedures. The client application embeds SQL statements and communicates with the server separately for each statement. This application design results in increased network traffic and processor costs.

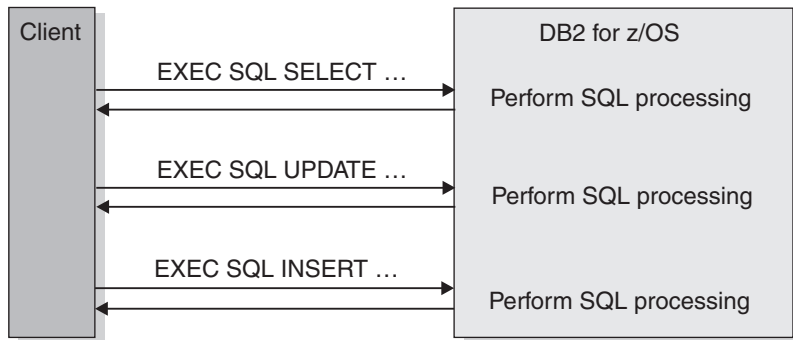


Figure 28. Processing without stored procedures

The following diagram illustrates the processing for an application that uses stored procedures. Because a stored procedure is used on the server, a series of SQL statements can be executed with a single send and receive operation, reducing network traffic and the cost of processing these statements.

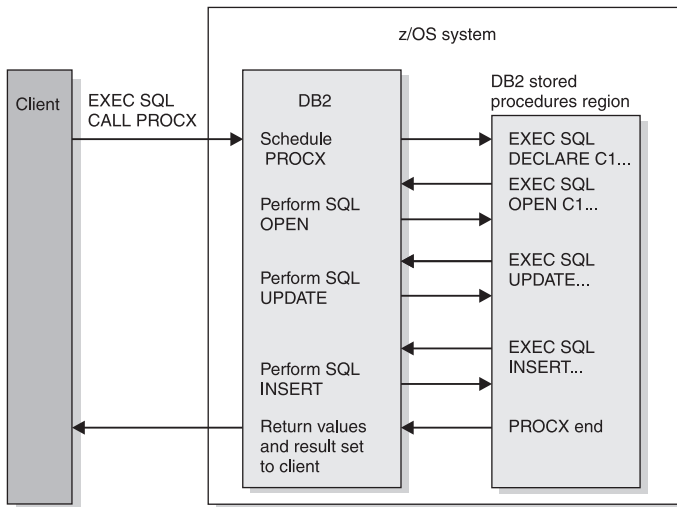


Figure 29. Processing with stored procedures

Stored procedures are useful for client/server applications that do at least one of the following things:

- Execute multiple remote SQL statements. Remote SQL statements can create many network send and receive operations, which results in increased processor costs. Stored procedures can encapsulate many of your application's SQL statements into a single message to the DB2 server, reducing network traffic to a single send and receive operation for a series of SQL statements. Locks on DB2 tables are not held across network transmissions, which reduces contention for resources at the server.
- Access tables from a dynamic SQL environment where table privileges for the application that is running are undesirable. Stored procedures allow static SQL authorization from a dynamic environment.
- Access host variables for which you want to guarantee security and integrity. Stored procedures remove SQL applications from the workstation, which prevents workstation users from manipulating the contents of sensitive SQL statements and host variables.
- Create a result set of rows to return to the client application.

Stored procedures that are written in embedded static SQL provide the following additional advantages:

- Better performance because static SQL is prepared at precompile time and has no run time overhead for access plan (package) generation.
- Encapsulation enables programmers to write applications that access data without knowing the details of database objects.
- Improved security because access privileges are encapsulated within the packages that are associated with the stored procedures. You can grant access to run a stored procedure that selects data from tables, without granting SELECT privilege to the user.

You can create one of the following types of stored procedures:

External stored procedures

A procedure that is written in a host language.

External SQL procedures

A procedure whose body is written entirely in SQL, but is created, implemented, and executed like other external stored procedures.

Native SQL procedures

A procedure with a procedural body that is written entirely in SQL and is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not have an associated external application program.

DB2 also provides a set of stored procedures that you can call in your application programs to perform a number of utility, application programming, and performance management functions. These procedures are called *DB2-supplied stored procedures*. Typically, you create these procedures during installation or migration.

Stored procedure parameters

You can pass information between a stored procedure and the calling application program by using parameters. Applications pass the required parameters in the SQL CALL statement. Optionally, the application can also include an indicator variable with each parameter to allow for null values or to pass large output parameter values.

You define the stored procedure parameters as part of the stored procedure definition in the CREATE PROCEDURE statement. The stored procedure parameters can be one of the following types:

IN Input-only parameters, which provide values to the stored procedure.

OUT Output-only parameters, which return values from the stored procedure to the calling program.

INOUT

Input and output parameters, which provide values to and return values from the stored procedure.

If a stored procedure fails to set one or more of the OUT or INOUT parameters, DB2 does not return an error. Instead, DB2 returns the output parameters to the calling program, with the values that were established on entry to the stored procedure.

Within a procedure body, the following rules apply to IN, OUT, and INOUT parameters:

- You can use a parameter that you define as IN on the left side or right side of an assignment statement. However, if you assign a value to an IN parameter, you cannot pass the new value back to the caller. The IN parameter has the same value before and after the SQL procedure is called.
- You can use a parameter that you define as OUT on the left side or right side of an assignment statement. The last value that you assign to the parameter is the value that is returned to the caller. The starting value of an OUT parameter is NULL.
- You can use a parameter that you define as INOUT on the left side or right side of an assignment statement. The caller determines the first value of the INOUT parameter, and the last value that you assign to the parameter is the value that is returned to the caller.

Restrictions:

- You cannot pass file reference variables as stored procedure parameters.

- You cannot pass parameters with the type XML to stored procedures. You can specify tables or views that contain XML columns as table locator parameters. However, you cannot reference the XML columns in the body of the stored procedure.

Related tasks:

Chapter 14, “Calling a stored procedure from your application,” on page 791

“Passing large output parameters to stored procedures by using indicator variables” on page 796

Related reference:

 CALL (DB2 SQL)

 CREATE PROCEDURE (DB2 SQL)

Example of a simple stored procedure

When an application that runs on a workstation calls a stored procedure on a DB2 server, the stored procedure updates a table based on the information that it receives from the application.

Suppose that an application runs on a workstation client and calls a stored procedure A on the DB2 server at location LOCA. Stored procedure A performs the following operations:

1. Receives a set of parameters containing the data for one row of the employee to project activity table (DSN8A10.EMPPROJECT). These parameters are input parameters in the SQL statement CALL:
 - EMP: employee number
 - PRJ: project number
 - ACT: activity ID
 - EMT: percent of employee's time required
 - EMS: date the activity starts
 - EME: date the activity is due to end
2. Declares a cursor, C1, with the option WITH RETURN, that is used to return a result set containing all rows in EMPPROJECT to the workstation application that called the stored procedure.
3. Queries table EMPPROJECT to determine whether a row exists where columns PROJNO, ACTNO, EMSTDATE, and EMPNO match the values of parameters PRJ, ACT, EMS, and EMP. (The table has a unique index on those columns. There is at most one row with those values.)
4. If the row exists, executes an SQL statement UPDATE to assign the values of parameters EMT and EME to columns EMPTIME and EMENDATE.¹
5. If the row does not exist (SQLCODE +100), executes an SQL statement INSERT to insert a new row with all the values in the parameter list.¹
6. Opens cursor C1. This causes the result set to be returned to the caller when the stored procedure ends.
7. Returns two parameters, containing these values:
 - A code to identify the type of SQL statement last executed: UPDATE or INSERT.
 - The SQLCODE from that statement.

Note:

1. Alternatively, steps 4 and 5 can be accomplished with a single MERGE statement.

The following figure illustrates the steps that are involved in executing this stored procedure.

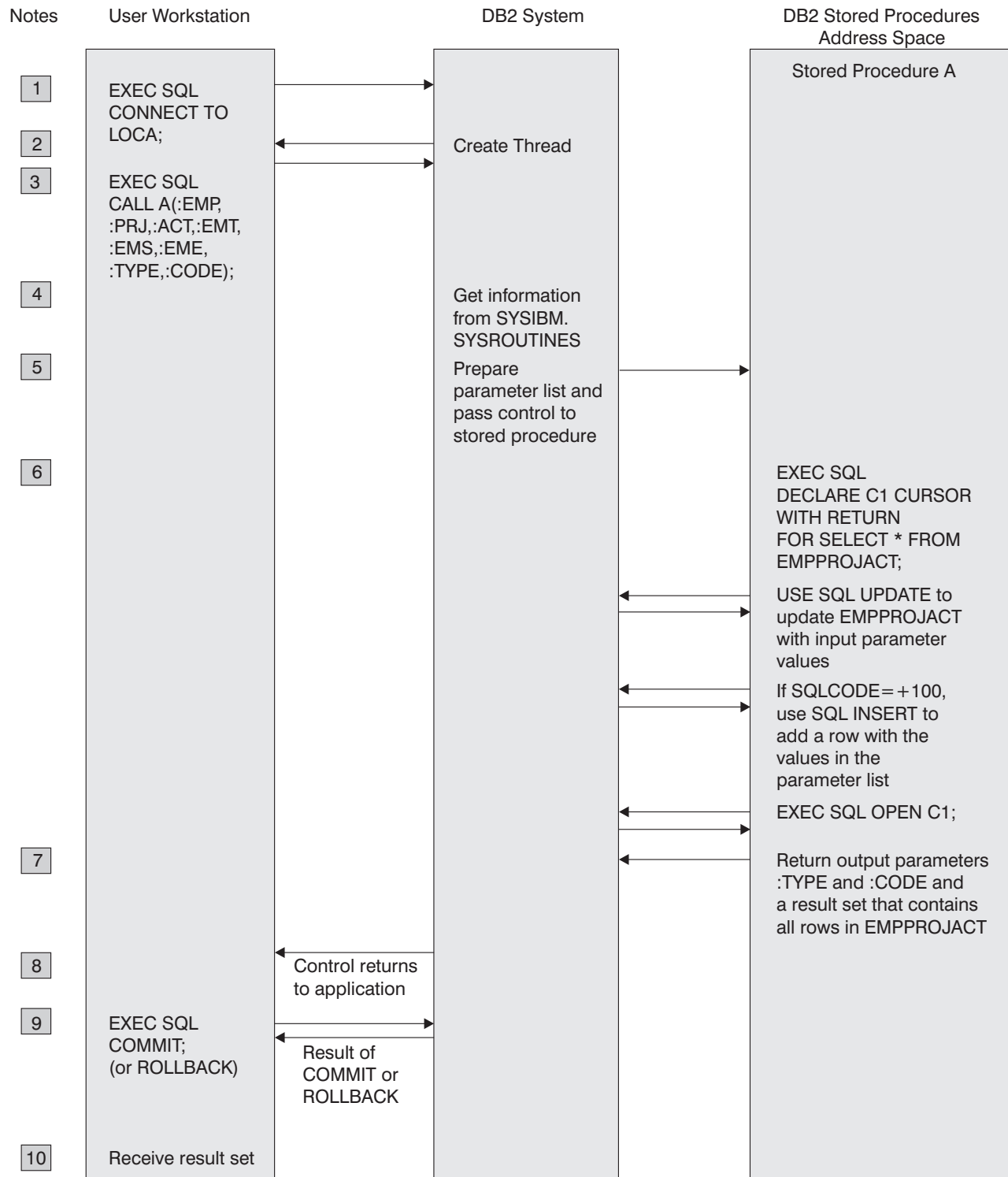


Figure 30. Stored procedure overview

Notes:

1. The workstation application uses the SQL CONNECT statement to create a conversation with DB2.

2. DB2 creates a DB2 thread to process SQL requests.
3. The SQL statement CALL tells the DB2 server that the application is going to run a stored procedure. The calling application provides the necessary parameters.
4. The plan for the client application contains information from catalog table SYSIBM.SYSROUTINES about stored procedure A.
5. DB2 passes information about the request to the stored procedures address space, and the stored procedure begins execution.
6. The stored procedure executes SQL statements.

DB2 verifies that the owner of the package or plan containing the SQL statement CALL has EXECUTE authority for the package associated with the DB2 stored procedure.

One of the SQL statements opens a cursor that has been declared WITH RETURN. This causes a result set to be returned to the workstation application when the procedure ends.

Any SQLCODE that is issued within an **external** stored procedure is **not** returned to the workstation application in the SQLCA (as the result of the CALL statement).

7. If an error is not encountered, the stored procedure assigns values to the output parameters and exits.

Control returns to the DB2 stored procedures address space, and from there to the DB2 system. If the stored procedure definition contains COMMIT ON RETURN NO, DB2 does not commit or roll back any changes from the SQL in the stored procedure until the calling program executes an explicit COMMIT or ROLLBACK statement. If the stored procedure definition contains COMMIT ON RETURN YES, and the stored procedure executed successfully, DB2 commits all changes. The COMMIT statement closes the cursor unless it is declared with the WITH HOLD option.

8. Control returns to the calling application, which receives the output parameters and the result set. DB2 then:

- Closes all cursors that the stored procedure opened, except those that the stored procedure opened to return result sets.
- Discards all SQL statements that the stored procedure prepared.
- Reclaims the working storage that the stored procedure used.

The application can call more stored procedures, or it can execute more SQL statements. DB2 receives and processes the COMMIT or ROLLBACK request. The COMMIT or ROLLBACK operation covers all SQL operations, whether executed by the application or by stored procedures, for that unit of work.

If the application involves IMS or CICS, similar processing occurs based on the IMS or CICS sync point rather than on an SQL COMMIT or ROLLBACK statement.

9. DB2 returns a reply message to the application describing the outcome of the COMMIT or ROLLBACK operation.
10. The workstation application executes the following steps to retrieve the contents of table EMPPROJACT, which the stored procedure has returned in a result set:
 - a. Declares a result set locator for the result set being returned.
 - b. Executes the ASSOCIATE LOCATORS statement to associate the result set locator with the result set.
 - c. Executes the ALLOCATE CURSOR statement to associate a cursor with the result set.

- d. Executes the FETCH statement with the allocated cursor multiple times to retrieve the rows in the result set.
- e. Executes the CLOSE statement to close the cursor.

SQL procedures

An SQL procedure is a stored procedure that contains only SQL statements.

The source code for these procedures (the SQL statements) is specified in an SQL CREATE PROCEDURE statement. The part of the CREATE PROCEDURE statement that contains SQL statements is called the *procedure body*.

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

Native SQL procedures

A procedure with a procedural body that is written entirely in SQL and is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not have an associated external application program.

External SQL procedures

A procedure whose body is written entirely in SQL, but is created, implemented, and executed like other external stored procedures.

Native SQL procedures

A *native SQL procedure* is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language. A native SQL procedure is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not require any other program preparation, such as precompiling, compiling, or link-editing source code. Native SQL procedures are executed as SQL statements that are bound in a DB2 package. Native SQL procedures do not have an associated external application program.

Native SQL procedures have the following advantages:

- You can create them in one step.
- They do not run in a WLM environment.
- They might be eligible for zIIP redirect if they are invoked remotely through a DRDA client.
- They usually perform better than external SQL procedures.
- They support more capabilities, such as nested compound statements, than external SQL procedures.
- DB2 can manage multiple versions of these procedures for you.

Starting in Version 9.1, all SQL procedures that are created without the FENCED or EXTERNAL options in the CREATE PROCEDURE statement are native SQL procedures.

External SQL procedures

An *external stored procedure* is a procedure that is written in a host language. An external stored procedure is much like any other SQL application. It can include static or dynamic SQL statements, IFI calls, and DB2 commands that are issued through IFI. You prepare external stored procedures as you would normally prepare application programs. You precompile, compile, and link-edit them. Then, you bind the DBRM into a package. You also need to define the procedure to DB2 by using the CREATE PROCEDURE statement. Thus, the source code for an external stored procedure is separate from the definition for the stored procedure.

All SQL procedures that were created prior to Version 9.1 are external SQL procedures. Starting in Version 9.1, you can create an external SQL procedure by specifying `FENCED` or `EXTERNAL` in the `CREATE PROCEDURE` statement.

SQL procedure body:

The body of an SQL procedure contains one or more SQL statements. In the SQL procedure body, you can also declare variables, condition handlers, reference parameters, and reference variables.

Statements that you can include in an SQL procedure body

An SQL procedure consists of a single SQL procedure statement. That procedure statement can be either an SQL control statement or another SQL statement. If the SQL control statement is a compound statement or a `CASE` statement, the procedure body can contain multiple statements. For native SQL procedures, you can use nested compound statements.

How to code multiple statements in an SQL procedure

Use a semicolon character to separate SQL statements within an SQL procedure.

The procedure body has no terminating character. Therefore, if the procedure contains only one statement, you do not need to put a semicolon after that statement. If the procedure consists of a set of nested statements, you do not need to put a semicolon after the outermost statement.

Variables in an SQL procedure

To store data that you use only within an SQL procedure, you can declare *SQL variables*. SQL variables are the equivalent of host variables in external stored procedures. SQL variables can have the same data types and lengths as SQL procedure parameters.

An SQL variable declaration has the following form:

```
DECLARE SQL-variable-name data-type;
```

The declaration for an SQL variable for which you use a result locator has the following form:

```
DECLARE SQL-variable-name data-type RESULT_SET_LOCATOR VARYING;
```

SQL variables in SQL procedures are subject to the following rules:

- SQL variable names, condition names, and label names must be less than or equal to 128 bytes in length. The names can include alphanumeric characters and the underscore character.
- SQL variable names must be unique. You cannot declare two SQL variables that have the same name, regardless of case. For example, you cannot declare two SQL variables named `varx` and `VARX`. (DB2 treats all SQL variable names as uppercase.)
- SQL parameters, SQL variables, and SQL conditions should not include SQL reserved words. Although doing so is not recommended, you can specify an SQL reserved word as the name of an SQL parameter, SQL variable, or SQL condition in some contexts. If you specify a reserved word as the name of an SQL parameter, SQL variable, or SQL condition in a context where its use could be ambiguous, specify the name as a delimited identifier.

- When you use an SQL variable in an SQL statement, do not precede the variable with a colon.

You can perform any operations on SQL variables that you can perform on host variables in SQL statements.

Object references in an SQL procedure

To avoid ambiguity, qualify SQL variable names and other object names. Use the following guidelines to determine when to qualify object names:

- Qualify column names with the associated table names or view names.
- When you use an SQL procedure parameter in the procedure body, qualify the parameter name with the procedure name.
- Specify a label for each compound statement, and qualify all SQL variables with the label name of the compound statement that declared them.

Calls to user-defined functions from an SQL procedure

When you call a user-defined function from an SQL procedure, ensure that you pass parameters of the appropriate data type. The data type should be the same data type or a data type that can be promoted to the data type of the function definition. For example, DB2 can promote the data type CHAR to VARCHAR or SMALLINT to BIGINT.

Related concepts:

“Nested compound statements in native SQL procedures” on page 575

“Stored procedure parameters” on page 561

 Promotion of data types (DB2 SQL)

 SQL control statements for external SQL procedures (DB2 SQL)

 SQL control statements for SQL routines (DB2 SQL)

Related reference:

 SQL-procedure-statement (DB2 SQL)

Examples of SQL procedures:

You can use CASE statements, compound statements, and nested statements within an SQL procedure body.

Example: CASE statement: The following SQL procedure demonstrates how to use a CASE statement. The procedure receives an employee's ID number and rating as input parameters. The CASE statement modifies the employee's salary and bonus, using a different UPDATE statement for each of the possible ratings.

```
CREATE PROCEDURE UPDATESALARY2
  (IN EMPNUMBR CHAR(6),
   IN RATING INT)
LANGUAGE SQL
MODIFIES SQL DATA
CASE RATING
WHEN 1 THEN
  UPDATE CORPDATA.EMPLOYEE
  SET SALARY = SALARY * 1.10, BONUS = 1000
  WHERE EMPNO = EMPNUMBR;
WHEN 2 THEN
  UPDATE CORPDATA.EMPLOYEE
  SET SALARY = SALARY * 1.05, BONUS = 500
```

```

        WHERE EMPNO = EMPNUMBR;
ELSE
    UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.03, BONUS = 0
        WHERE EMPNO = EMPNUMBR;
END CASE

```

Example: Compound statement with nested IF and WHILE statements: The following example shows a compound statement that includes an IF statement, a WHILE statement, and assignment statements. The example also shows how to declare SQL variables, cursors, and handlers for classes of error codes.

The procedure receives a department number as an input parameter. A WHILE statement in the procedure body fetches the salary and bonus for each employee in the department, and uses an SQL variable to calculate a running total of employee salaries for the department. An IF statement within the WHILE statement tests for positive bonuses and increments an SQL variable that counts the number of bonuses in the department. When all employee records in the department have been processed, a NOT FOUND condition occurs. A NOT FOUND condition handler makes the search condition for the WHILE statement false, so execution of the WHILE statement ends. Assignment statements then assign the total employee salaries and the number of bonuses for the department to the output parameters for the stored procedure.

If any SQL statement in the compound statement P1 receives an error, the SQLEXCEPTION handler receives control. The handler action sets the output parameter DEPTSALARY to NULL. After the handler action has completed successfully, the original error condition is resolved (SQLSTATE '00000', SQLCODE 0). Because this handler is an EXIT handler, execution passes to the end of the compound statement, and the SQL procedure ends.

```

CREATE PROCEDURE RETURNDEPTSALARY
    (IN DEPTNUMBER CHAR(3),
     OUT DEPTSALARY DECIMAL(15,2),
     OUT DEPTBONUSCNT INT)
LANGUAGE SQL
READS SQL DATA
P1: BEGIN
    DECLARE EMPLOYEE_SALARY DECIMAL(9,2);
    DECLARE EMPLOYEE_BONUS DECIMAL(9,2);
    DECLARE TOTAL_SALARY DECIMAL(15,2) DEFAULT 0;
    DECLARE BONUS_CNT INT DEFAULT 0;
    DECLARE END_TABLE INT DEFAULT 0;
    DECLARE C1 CURSOR FOR
        SELECT SALARY, BONUS FROM CORPDATA.EMPLOYEE
            WHERE WORKDEPT = DEPTNUMBER;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET END_TABLE = 1;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        SET DEPTSALARY = NULL;
    OPEN C1;
    FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
    WHILE END_TABLE = 0 DO
        SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
        IF EMPLOYEE_BONUS > 0 THEN
            SET BONUS_CNT = BONUS_CNT + 1;
        END IF;
        FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
    END WHILE;
    CLOSE C1;
    SET DEPTSALARY = TOTAL_SALARY;
    SET DEPTBONUSCNT = BONUS_CNT;
END P1

```

Example: Compound statement with dynamic SQL statements: The following example shows a compound statement that includes dynamic SQL statements.

The procedure receives a department number (P_DEPT) as an input parameter. In the compound statement, three statement strings are built, prepared, and executed:

- The first statement string executes a DROP statement to ensure that the table to be created does not already exist. This table is named DEPT_*deptno*_T, where *deptno* is the value of input parameter P_DEPT.
- The next statement string executes a CREATE statement to create DEPT_*deptno*_T.
- The third statement string inserts rows for employees in department *deptno* into DEPT_*deptno*_T.

Just as statement strings that are prepared in host language programs cannot contain host variables, statement strings in SQL procedures cannot contain SQL variables or stored procedure parameters. Therefore, the third statement string contains a parameter marker that represents P_DEPT. When the prepared statement is executed, parameter P_DEPT is substituted for the parameter marker.

```
CREATE PROCEDURE CREATEDEPTTABLE (IN P_DEPT CHAR(3))
LANGUAGE SQL
BEGIN
  DECLARE STMT CHAR(1000);
  DECLARE MESSAGE CHAR(20);
  DECLARE TABLE_NAME CHAR(30);
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET MESSAGE = 'ok';
  SET TABLE_NAME = 'DEPT_' || P_DEPT || '_T';
  SET STMT = 'DROP TABLE ' || TABLE_NAME;
  PREPARE S1 FROM STMT;
  EXECUTE S1;
  SET STMT = 'CREATE TABLE ' || TABLE_NAME ||
    '( EMPNO CHAR(6) NOT NULL, ' ||
    'FIRSTNME VARCHAR(6) NOT NULL, ' ||
    'MIDINIT CHAR(1) NOT NULL, ' ||
    'LASTNAME CHAR(15) NOT NULL, ' ||
    'SALARY DECIMAL(9,2))';
  PREPARE S2 FROM STMT;
  EXECUTE S2;
  SET STMT = 'INSERT INTO ' || TABLE_NAME ||
    'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY ' ||
    'FROM EMPLOYEE ' ||
    'WHERE WORKDEPT = ?';
  PREPARE S3 FROM STMT;
  EXECUTE S3 USING P_DEPT;
END
```

External stored procedures

An *external stored procedure* is a procedure that is written in a host language and can contain SQL statements. The source code for external procedures is separate from the definition.

An external stored procedure is much like any other SQL application. It can include static or dynamic SQL statements, IFI calls, and DB2 commands that are issued through IFI. You prepare external stored procedures as you would normally prepare application programs. You precompile, compile, and link-edit them. Then, you bind the DBRM into a package. You also need to define the procedure to DB2 by using the CREATE PROCEDURE statement. Thus, the source code for an external stored procedure is separate from the definition for the stored procedure.

Language requirements for the external stored procedure and its caller

You can write an external stored procedure in Assembler, C, C++, COBOL, Java, REXX, or PL/I. All programs must be designed to run using Language Environment. Your COBOL and C++ stored procedures can contain object-oriented extensions.

The program that calls the stored procedure can be in any language that supports the SQL CALL statement. ODBC applications can use an escape clause to pass a stored procedure call to DB2.

Related concepts:

“Object-oriented extensions in COBOL” on page 363

“REXX stored procedures” on page 650

 Java stored procedures and user-defined functions (DB2 Application Programming for Java)

Differences between SQL procedures and external procedures

SQL procedures are written entirely in SQL statements. External procedures are written in a host language and can contain SQL statements. You can invoke both types of procedures with an SQL CALL statement. However, you should consider several important differences in behavior and preparation.

SQL procedures and external procedures consist of a procedure definition and the code for the procedure program.

Both an SQL procedure definition and an external procedure definition specify the following information:

- The procedure name.
- Input and output parameter attributes.
- The language in which the procedure is written. For an SQL procedure, the language is SQL.
- Information that will be used when the procedure is called, such as run time options, length of time that the procedure can run, and whether the procedure returns result sets.

An SQL procedure and external procedure share the same rules for the use of COMMIT and ROLLBACK statements in a procedure.

An SQL procedure and an external procedure differ in the following ways:

- How they handle errors:
 - For an SQL procedure, DB2 automatically returns SQL conditions in the SQLCA when the procedure does not include a RETURN statement or a handler. For information about the various ways to handle errors in an SQL procedure, see “Handling SQL conditions in an SQL procedure” on page 578.
 - For an external stored procedure, DB2 does **not** return SQL conditions in the SQLCA to the workstation application. If you use PARAMETER STYLE SQL when you define an external procedure, you can set SQLSTATE to indicate an error before the procedure ends. For valid SQLSTATE values, see “Parameters for external user-defined functions” on page 528.
- How they specify the code for the stored procedure. An SQL procedure definition contains the source code for the stored procedure. An external stored procedure definition specifies the name of the stored procedure program.

- How you define the stored procedure. For native SQL procedures and external procedures, you define the stored procedure to DB2 by executing the CREATE PROCEDURE statement. For external SQL procedures, you define the stored procedure to DB2 by preprocessing a CREATE PROCEDURE statement, then executing the CREATE PROCEDURE statement dynamically. For all procedures, you change the definition by executing the ALTER PROCEDURE statement. See “Creating an external SQL procedure” on page 600 for more information about defining an SQL procedure to DB2.

Example: The following example shows a definition for an SQL procedure.

```
CREATE PROCEDURE UPDATESALARY1      1
  (IN EMPNUMBR CHAR(10),           2
  IN RATE DECIMAL(6,2))
LANGUAGE SQL                       3
UPDATE EMP                         4
  SET SALARY = SALARY * RATE
  WHERE EMPNO = EMPNUMBR
```

Notes:

- 1 The stored procedure name is UPDATESALARY1.
- 2 The two parameters have data types of CHAR(10) and DECIMAL(6,2). Both are input parameters.
- 3 LANGUAGE SQL indicates that this is an SQL procedure, so a procedure body follows the other parameters.
- 4 The procedure body consists of a single SQL UPDATE statement, which updates rows in the employee table.

Example: The following example shows a definition for an equivalent external stored procedure that is written in COBOL. The stored procedure program, which updates employee salaries, is called UPDSAL.

```
CREATE PROCEDURE UPDATESALARY1      1
  (IN EMPNUMBR CHAR(10),           2
  IN RATE DECIMAL(6,2))
LANGUAGE COBOL                     3
EXTERNAL NAME UPDSAL;              4
```

Notes:

- 1 The stored procedure name is UPDATESALARY1.
- 2 The two parameters have data types of CHAR(10) and DECIMAL(6,2). Both are input parameters.
- 3 LANGUAGE COBOL indicates that this is an external procedure, so the code for the stored procedure is in a separate, COBOL program.
- 4 The name of the load module that contains the executable stored procedure program is UPDSAL.

COMMIT and ROLLBACK statements in a stored procedure

When you issue COMMIT or ROLLBACK statements in your stored procedure, DB2 commits or rolls back all changes within the unit of work. These changes include changes that the client application made before it called the stored procedure and DB2 work that the stored procedure does.

If your stored procedure includes COMMIT or ROLLBACK statements, define it with the one of the following clauses:

- CONTAINS SQL

- READS SQL DATA
- MODIFIES SQL DATA

The COMMIT ON RETURN clause in a stored procedure definition has no effect on the COMMIT or ROLLBACK statements in the stored procedure code. If you specify COMMIT ON RETURN YES when you define the stored procedure, DB2 issues a COMMIT statement when control returns from the stored procedure. This action occurs regardless of whether the stored procedure contains COMMIT or ROLLBACK statements.


A ROLLBACK statement has the same effect on cursors in a stored procedure as it has on cursors in stand-alone programs. A ROLLBACK statement closes all open cursors. A COMMIT statement in a stored procedure closes cursors that are not declared WITH HOLD and leaves open those cursors that are declared WITH HOLD. The effect of COMMIT or ROLLBACK on cursors applies to cursors that are declared in the calling application and to cursors that are declared in the stored procedure.

Restriction: You cannot include COMMIT or ROLLBACK statements in a stored procedure if any of the following conditions are true:

- The stored procedure is nested within a trigger or user-defined function.
- The stored procedure is called by a client that uses two-phase commit processing.
- The client program uses a type 2 connection to connect to the remote server that contains the stored procedure.
- DB2 is not the commit coordinator.

If a COMMIT or ROLLBACK statement in a stored procedure violates any of these conditions, DB2 puts the transaction in a must-rollback state. Also, in this case, the CALL statement fails.

Related reference:

 [CALL \(DB2 SQL\)](#)

 [COMMIT \(DB2 SQL\)](#)

 [ROLLBACK \(DB2 SQL\)](#)

Special registers in a stored procedure

You can use all special registers in a stored procedure. However, you can modify only some of those special registers. After a stored procedure completes, DB2 restores all special registers to the values that they had before invocation.

Creating a native SQL procedure

A *native SQL procedure* is a procedure whose body is written entirely in SQL and is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures typically perform better and have more functionality than external SQL procedures.

Before you create a native SQL procedure, Configure DB2 for running stored procedures and user-defined functions during installation or Configure DB2 for running stored procedures and user-defined functions during migration (DB2 Installation Guide) if the native SQL procedure satisfies at least one of the following conditions:

- The native SQL procedure calls at least one external stored procedure, external SQL procedure, or user-defined function.

- The native SQL procedure is defined with ALLOW DEBUG MODE or DISALLOW DEBUG MODE. If you specify DISABLE DEBUG MODE, you do not need to set up the stored procedure environment.

A *native SQL procedure* is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language. A native SQL procedure is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not require any other program preparation, such as precompiling, compiling, or link-editing source code. Native SQL procedures are executed as SQL statements that are bound in a DB2 package. Native SQL procedures do not have an associated external application program.

To create a native SQL procedure, perform one of the following actions:

- Use IBM Data Studio to specify the source statements for the SQL procedure and deploy the SQL procedure to DB2. IBM Data Studio also allows you to create copies of the procedure package as needed and to deploy the procedure to remote servers.
- Manually deploy the native SQL procedure by completing the following steps:
 1. Issue the CREATE PROCEDURE statement:
 - Include the procedure body, which is written entirely in SQL, in the SQL procedural language. For more information about what you can do within the procedure body, see the following information:
 - “Controlling the scope of variables in an SQL procedure” on page 574
 - “Declaring cursors in an SQL procedure with nested compound statements” on page 577
 - “Handling SQL conditions in an SQL procedure” on page 578
 - “Raising a condition within an SQL procedure by using the SIGNAL or RESIGNAL statements” on page 588
 - Do not include the FENCED or EXTERNAL keywords.

When you issue this CREATE PROCEDURE statement, the first version of this procedure is defined to DB2, and a package is implicitly bound with the options that you specify on the CREATE PROCEDURE statement.

2. If the native SQL procedure contains one or more of the following statements or references, make copies of the native SQL procedure package, as needed:
 - CONNECT
 - SET CURRENT PACKAGESET
 - SET CURRENT PACKAGE PATH
 - A table reference with a three-part name
3. If you plan to call the native SQL procedure at another DB2 server, deploy the procedure to another DB2 for z/OS server. You can customize the bind options at the same time.
4. Authorize the appropriate users to call the stored procedure.

After you create a native SQL procedure, you can create one or more versions of it as needed.

Related concepts:

“SQL procedures” on page 565

“SQL procedure body” on page 566

Related tasks:

 [Implementing DB2 stored procedures \(DB2 Administration Guide\)](#)

➤ Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio)

Related reference:

➤ CREATE PROCEDURE (SQL - native) (DB2 SQL)

Controlling the scope of variables in an SQL procedure

Use nested compound statements within an SQL procedure to define the scope of SQL variables. You can reference the variable only within the compound statement in which it was declared and within any nested statements.

To control the scope of a variable in an SQL procedure:

1. Declare the variable within the compound statement in which you want to reference it. Ensure that the variable name is unique within the compound statement, not including any nested statements. You can define variables with the same name in other compound statements in the same SQL procedure.
2. Reference the variable within that compound statement or any nested statements.

Recommendation: If multiple variables with the same name exist within an SQL procedure, qualify the variable with the label from the compound statement in which it was declared. Otherwise, you might accidentally reference the wrong variable.

If the variable name is unqualified and multiple variables with that name exist within the same scope, DB2 uses the variable in the innermost compound statement.

Example: The following example contains three declarations of the variable A. One instance is declared in the outer compound statement, which has the label OUTER1. The other instances are declared in the inner compound statements with the labels INNER1 and INNER2. In the INNER1 compound statement, DB2 presumes that the unqualified references to A in the assignment statement and UPDATE statement refer to the instance of A that is declared in the INNER1 compound statement. To refer to the instance of A that is declared in the OUTER1 compound statement, qualify the variable as OUTER1.A.

```
CREATE PROCEDURE P2 ()
    LANGUAGE SQL

    -- Outermost compound statement -----
    OUTER1: BEGIN 1
        DECLARE A INT DEFAULT 100;

        -- Inner compound statement with label INNER1 ---
        INNER1: BEGIN 2
            DECLARE A INT DEFAULT NULL;
            DECLARE W INT DEFAULT NULL;

            SET A = A + OUTER1.A; 3

            UPDATE T1 SET T1.B = 5
                WHERE T1.B = A; 4

            SET OUTER1.A = 100; 5

            SET INNER1.A = 200; 6
        END INNER1; 7
    -- End of inner compound statement INNER1 -----

    -- Inner compound statement with label INNER2 ---
```



```

INNER2: BEGIN 8
    DECLARE A INT DEFAULT NULL;
    DECLARE Z INT DEFAULT NULL;

    SET A = A + OUTER1.A;

END INNER2; 9
-- End of inner compound statement INNER2 -----

SET OUTER1.A = 100; 10

END OUTER1 11

```

The preceding example has the following parts:

1. The beginning of the outermost compound statement, which has the label OUTER1.
2. The beginning of the inner compound statement with the label INNER1.
3. The unqualified variable A refers to INNER1.A.
4. The unqualified variable A refers to INNER1.A.
5. OUTER1.A is a valid reference, because this variable is referenced in a nested compound statement.
6. INNER1.A is a valid reference, because this variable is referenced in the same compound statement in which it is declared. You cannot reference INNER2.A, because this variable is not in the scope of this compound statement.
7. The end of the inner compound statement with the label INNER1.
8. The beginning of the inner compound statement with the label INNER2.
9. The end of the inner compound statement with the label INNER2.
10. OUTER1.A is a valid reference, because this variable is referenced in the same compound statement in which it is declared. You cannot reference INNER1.A, because this variable is declared in a nested statement and cannot be referenced in the outer statement.
11. The end of the outermost compound statement, which has the label OUTER1.

Nested compound statements in native SQL procedures:

Nested compound statements are blocks of SQL statements that are contained by other blocks of SQL statements in native SQL procedures. Use nested compound statements to define condition handlers that execute more than one statement and to define different scopes for variables and condition handlers.

The following pseudo code shows a basic structure of an SQL procedure with nested compound statements:

```

CREATE PROCEDURE...
  OUTERMOST: BEGIN
    ...
    INNER1: BEGIN
      ...
      INNERMOST: BEGIN
        ...
        ...
        END INNERMOST;
      END INNER1;
    INNER2: BEGIN
      ...
      ...
      END INNER2;
  END OUTERMOST

```

In the preceding code, the OUTERMOST compound statement contains two nested compound statements: INNER1 and INNER2. INNER1 contains one nested compound statement: INNERMOST.

Related concepts:

“Handlers in an SQL procedure” on page 578

Related tasks:

“Defining condition handlers that execute more than one statement” on page 579

Statement labels for nested compound statements in native SQL procedures:

You can define a label for each compound statement in an SQL procedure. This label enables you to reference this block of statements in other statements such as the GOTO, LEAVE, and ITERATE SQL PL control statements. You can also use the label to qualify a variable when necessary. Labels are not required.

A label name must meet the following criteria:

- Be unique within the compound statement, including any compound statements that are nested within the compound statement.
- Not be the same as the name of the SQL procedure.

You can reference a label within the compound statement in which it is defined, including any compound statements that are nested within that compound statement.

Example of statement labels: The following example shows several statement labels and their scope:

```
CREATE PROCEDURE P1 ()
    LANGUAGE SQL

    --Outermost compound statement -----
    OUTER1: BEGIN 1

        --Inner compound statement with label INNER1 ---
        INNER1: BEGIN 2
            IF...
                ABC: LEAVE INNER1; 3
            ELSEIF
                XYZ: LEAVE OUTER1; 4
            END IF

        END INNER1;
    --End of inner compound statement INNER1 -----

    --Inner compound statement with label INNER2---
    INNER2: BEGIN 5
        XYZ:...statement 6
    END INNER2;
    -- End of inner compound statement INNER2 -----

END OUTER1 7
```

The preceding example has the following parts:

1. The beginning of the outermost compound statement, which is labeled OUTER1
2. The beginning of an inner compound statement that is labeled INNER1
3. A LEAVE statement that is defined with the label ABC. This LEAVE statement specifies that DB2 is to terminate processing of the compound statement

- INNER1 and begin processing the next statement, which is INNER2. This LEAVE statement cannot specify INNER2, because that label is not within the scope of the INNER1 compound statement.
4. A LEAVE statement that is defined with the label XYZ. This LEAVE statement specifies that DB2 is to terminate processing of the compound statement OUTER1 and begin processing the next statement, if one exists. This example does not show the next statement.
 5. The beginning of an inner compound statement that is labeled INNER2.
 6. A statement that is defined with the label XYZ. This label is acceptable even though another statement in this procedure has the same label, because the two labels are in different scopes. Neither label is contained within the scope of the other.
 7. The end of the outermost compound statement that is labeled OUTER1.

The following examples show valid and invalid uses of labels:

Invalid example of labels:

```
L1: BEGIN
    L2: SET A = B;
    L1: GOTO L2: --This duplicate use of the label L1 causes an error, because
               --the same label is already used in the same scope.

END L1;
```

Valid example of labels:

```
L1: BEGIN
    L2: BEGIN
        L4: BEGIN --This line contains the first use of the label L4
            DECLARE A CHAR(5);
            SET A = B;
        END L4;
    END L2;

    L3: BEGIN
        L4: BEGIN --This second use of the label L4 is valid, because
            --it is used in a different scope.
            DECLARE A CHAR(5);
            SET A = B;
        END L4;
    END L3;
END L1;
```

Declaring cursors in an SQL procedure with nested compound statements

When you declare a cursor in an SQL procedure that has nested compound statements, you cannot necessarily reference the cursor anywhere in the procedure. The scope of the cursor is constrained to the compound statement in which you declare it.

To declare a cursor in an SQL procedure with nested compound statements:

Specify the DECLARE CURSOR statement within the compound statement in which you want to reference the cursor. Use a cursor name that is unique within the SQL procedure.

You can reference the cursor within the compound statement in which it is declared and within any nested statements. If the cursor is declared as a result set cursor, even if the cursor is not declared in the outermost compound statement, any calling application can reference it.

In the following example, cursor X is declared in the outer compound statement. This cursor can be referenced within the outer block in which it was declared and within any nested compound statements.

```
CREATE PROCEDURE SINGLE_CSR
  (INOUT IR1 INT, INOUT JR1 INT, INOUT IR2 INT, INOUT JR2 INT)
  LANGUAGE SQL
  DYNAMIC RESULT SETS 2
  BEGIN
    DECLARE I INT;
    DECLARE J INT;
    DECLARE X CURSOR WITH RETURN FOR --outer declaration for X
      SELECT * FROM CSRT1;

    SUB: BEGIN
      OPEN X;                                --references X in outer block
      FETCH X INTO I,J;                       --references X in outer block
      SET IR1 = I;
      SET JR1 = J;
    END;

    FETCH X INTO I,J;                         --references X in outer block
    SET IR2 = I;
    SET JR2 = j;
    CLOSE X;
  END
```

Related reference:

 [CREATE PROCEDURE \(SQL - native\) \(DB2 SQL\)](#)

 [DECLARE CURSOR \(DB2 SQL\)](#)

Handling SQL conditions in an SQL procedure

In an SQL procedure, you can specify how the program should handle certain SQL errors and SQL warnings.

If you do not include a handler or a RETURN statement in the SQL procedure, DB2 automatically returns any SQL conditions to the caller in the SQLCA.

To handle SQL conditions, use one of the following techniques:

- Include statements called *handlers* to tell the procedure to perform some other action when an error or warning occurs.
- Include a RETURN statement in an SQL procedure to return an integer status value to the caller.
- Include a SIGNAL statement or a RESIGNAL statement to raise a specific SQLSTATE and to define the message text for that SQLSTATE.
- Force a negative SQLCODE to be returned by a procedure if a trigger calls the procedure.

Handlers in an SQL procedure:

If an error occurs when an SQL procedure executes, the procedure ends unless you include statements to tell the procedure to perform some other action. These statements are called handlers.

Handlers are similar to WHENEVER statements in external SQL application programs. Handlers tell the SQL procedure what to do when an error or warning occurs, or when no more rows are returned from a query. In addition, you can

declare handlers for specific SQLSTATES. You can refer to an SQLSTATE by its number in a handler, or you can declare a name for the SQLSTATE and then use that name in the handler.

The general form of a handler declaration is:

```
DECLARE handler-type HANDLER FOR condition SQL-procedure-statement;
```

In general, the way that a handler works is that when an error occurs that matches *condition*, the *SQL-procedure-statement* executes. When the *SQL-procedure-statement* completes, DB2 performs the action that is indicated by *handler-type*.

Types of handlers

The handler type determines what happens after the completion of the *SQL-procedure-statement*. You can declare the handler type to be either CONTINUE or EXIT:

CONTINUE

Specifies that after *SQL-procedure-statement* completes, execution continues with the statement after the statement that caused the error.

EXIT

Specifies that after *SQL-procedure-statement* completes, execution continues at the end of the compound statement that contains the handler.

Example: CONTINUE handler: This handler sets flag `at_end` when no more rows satisfy a query. The handler then causes execution to continue after the statement that returned no rows.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET at_end=1;
```

Example: EXIT handler: This handler places the string 'Table does not exist' into output parameter `OUT_BUFFER` when condition `NO_TABLE` occurs. `NO_TABLE` is previously declared as SQLSTATE 42704 (*name* is an undefined name). The handler then causes the SQL procedure to exit the compound statement in which the handler is declared.

```
DECLARE NO_TABLE CONDITION FOR '42704';  
:  
:  
DECLARE EXIT HANDLER FOR NO_TABLE  
SET OUT_BUFFER='Table does not exist';
```

Defining condition handlers that execute more than one statement:

A *condition handler* defines the action that an SQL procedure takes when a particular condition occurs. You must specify the action as a single SQL procedure statement.

To define a condition handler that executes more than one statement when the specified condition occurs, specify a compound statement within the declaration of that handler.

Example: The following example shows a condition handler that captures the SQLSTATE value and sets a local flag to TRUE.

```
BEGIN  
  DECLARE SQLSTATE CHAR(5);  
  DECLARE PrvSQLState CHAR(5) DEFAULT '00000';  
  DECLARE ExceptState INT;  
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION  
  BEGIN
```

```

        SET PrvSQLState = SQLSTATE;
        SET ExceptState = TRUE;
    END;
    ...
END

```

Example: The following example declares a condition handler for SQLSTATE 72822. The subsequent SIGNAL statement is within the scope of this condition handler and thus activates this handler. The condition handler tests the value of the SQL variable VAR with an IF statement. Depending on the value of VAR, the SQLSTATE is changed and the message text is set.

```

DECLARE EXIT HANDLER FOR SQLSTATE '72822'
    IF ( VAR = 'OK' ) THEN
        RESIGNAL SQLSTATE '72623'
            SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
    ELSE
        RESIGNAL SQLSTATE '72319'
            SET MESSAGE_TEXT = VAR;
    END IF;

SIGNAL SQLSTATE '72822';

```

Controlling how errors are handled within different scopes in an SQL procedure:

You can use nested compound statements in an SQL procedure to specify that errors be handled differently within different scopes. You can also ensure that condition handlers are checked only with a particular compound statement.

To control how errors are handled within different scopes in an SQL procedure:

1. Optional: Declare a condition by specifying a DECLARE CONDITION statement within the compound statement in which you want to reference it. You can reference a condition in the declaration of a condition handler, a SIGNAL statement, or a RESIGNAL statement.

Restriction: If multiple conditions with that name exist within the same scope, you cannot explicitly refer to a condition that is not the most local in scope. DB2 uses the condition in the innermost compound statement.

2. Declare a condition handler by specifying a DECLARE HANDLER statement within the compound statement to which you want the condition handler to apply. Within the declaration of the condition handler, you can specify a previously defined condition.

Restriction: Condition handlers that are declared in the same compound statement cannot handle conditions encountered in each other or themselves.

Example: In the following example, a condition with the name ABC is declared twice, and a condition named XYZ is declared once.

```

CREATE PROCEDURE...
    DECLARE ABC CONDITION...

    DECLARE XYZ CONDITION...
    BEGIN
        DECLARE ABC CONDITION...
        SIGNAL ABC; 1
    END;

    SIGNAL ABC; 2

```

The following notes refer to the preceding example:

1. ABC refers to the condition that is declared in the innermost block. If this statement were changed to SIGNAL XYZ, XYZ would refer to the XYZ condition that is declared in the outermost block.
2. ABC refers to the condition that is declared in the outermost block.

Example: The following example contains multiple declarations of a condition with the name FOO, and a single declaration of a condition with the name GORP.

```
CREATE PROCEDURE MYTEST (INOUT A CHAR(1), INOUT B CHAR(1))
L1: BEGIN
  DECLARE GORP CONDITION
    FOR SQLSTATE '33333'; -- defines a condition with the name GORP for SQLSTATE 33333

  DECLARE EXIT HANDLER FOR GORP --defines a condition handler for SQLSTATE 33333
  L2: BEGIN
    DECLARE FOO CONDITION
      FOR SQLSTATE '12345'; --defines a condition with the name FOO for SQLSTATE 12345
    DECLARE CONTINUE HANDLER FOR FOO --defines a condition handler for SQLSTATE 12345
    L3: BEGIN
      SET A = 'A';
      ...more statements...
    END L3;
    SET B = 'B';

    IF...
      SIGNAL FOO; --raises SQLSTATE 12345
    ELSEIF
      SIGNAL GORP; --raises SQLSTATE 33333
    END IF;

  END L2;

L4: BEGIN
  DECLARE FOO CONDITION
    FOR SQLSTATE '54321' --defines a condition with the name FOO for SQLSTATE 54321
  DECLARE EXIT HANDLER FOR FOO...; --defines a condition handler for SQLSTATE 54321

  SIGNAL FOO SET MESSAGE_TEXT = '...'; --raises SQLSTATE 54321

  L5: BEGIN
    DECLARE FOO CONDITION
      FOR SQLSTATE '99999'; --defines a condition with the name FOO for SQLSTATE 99999
    ...more statements...
  END L5;

END L4;

--At this point, the procedure cannot reference FOO, because this condition is not defined
--in this outer scope

END L1
```

Example: In the following example, the compound statement with the label OUTER contains two other compound statements: INNER1A and INNER1B. The INNER1A compound statement contains another compound statement, which has the label INNER1A2, and the declaration for a condition handler HINNER1A. The body of the condition handler HINNER1A contains another compound statement, which defines another condition handler, HINNER1A_HANDLER.

```
OUTER:
  BEGIN
    -- Handler for OUTER
    DECLARE ... HANDLER -- HOUTER
    BEGIN
      <-----
      |
      <---.
```

```

:
END; -- End of handler <----.
:
:
-- Level 1 - first compound statement
INNER1A:
BEGIN <-----
  -- Handler for INNER1A
  DECLARE ... HANDLER -- HINNER1A
  BEGIN <-----
    -- Handler for handler HINNER1A
    DECLARE...HANDLER --HINNER1A_HANDLER
    BEGIN <----.
      :
      END; -- End of handler <----.
      :
      : -- stmt that gets condition
      :
      : -- more statements in handler
      END; -- End of HINNER1A handler<-----.

  INNER1A2:
  BEGIN <--
    DECLARE ... HANDLER...-- HINNER1A2
    BEGIN; <----.
      :
      END; -- End of handler <----.
      :
      : -- statement that gets condition
      :
      : -- statement after statement
      : -- that encountered condition
    END INNER1A2; <--
  :
  : -- statements in INNER1A
END INNER1A; <-----

-- Level 1 - second compound statement
INNER1B:
BEGIN <-----
  -- Handler for handler INNER1B
  DECLARE ...HANDLER -- HINNER1B
  BEGIN <-----
    -- Handler for HINNER1B --
    DECLARE ...HANDLER --HINNER1B_HANDLER
    BEGIN <----.
      :
      END; -- End of handler <----.
      :
      : -- statements in handler
      END; -- End of HINNER1B handler<-----.
  :
  : -- statements in INNER1B
END INNER1B; <-----

: -- statements in OUTER
END OUTER; <=====

```

2

1

The following notes apply to the preceding example:

1. If an exception, warning, or NOT FOUND condition occurs within the INNER1A2 compound statement, the most appropriate handler within that compound statement is activated to handle the condition. Then, one of the following actions occurs depending on the type of condition handler:

- If the condition handler (HINNER1A2) is an exit handler, control is returned to the end of the compound statement that contained the condition handler.
- If the condition handler (HINNER1A2) is a continue handler, processing continues with the statement after the statement that encountered the condition.

If no appropriate handler exists in the INNER1A2 compound statement, DB2 considers the following handlers in the specified order:

- a. The most appropriate handler within the INNER1A compound statement.
- b. The most appropriate handler within the OUTER compound statement.

If no appropriate handler exists in the OUTER compound statement, the condition is an unhandled condition. If the condition is an exception condition, the procedure terminates and returns an unhandled condition to the invoking application. If the condition is a warning or NOT FOUND condition, the procedure returns the unhandled warning condition to the invoking application.

2. If an exception, warning, or NOT FOUND condition occurs within the body of the condition handler HINNER1A, and the condition handler HINNER1A_HANDLER is the most appropriate handler for the exception, that handler is activated. Otherwise, the most appropriate handler within the OUTER compound statement handles the condition. If no appropriate handler exists within the OUTER compound statement, the condition is treated as an unhandled condition.

Example: In the following example, when *statement2* results in a NOT FOUND condition, the appropriate condition handler is activated to handle the condition. When the condition handler completes, the compound statement that contains that condition handler terminates, because the condition handler is an EXIT handler. Processing then continues with *statement4*.

```
BEGIN
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET OUT_OF_DATA_FLAG = ON;
  statement1...
  statement2... --assume that this statement results in a NOT FOUND condition
  statement3...
END;

statement4
...
```

Example: In the following example, DB2 checks for SQLSTATE 22H11 only for statements inside the INNER compound statement. DB2 checks for SQLEXCEPTION for all statements in both the OUTER and INNER blocks.

```
OUTER: BEGIN
  DECLARE var1 INT;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    RETURN -3;

  INNER: BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE '22H11'
      RETURN -1;
    DECLARE C1 CURSOR FOR SELECT col1 FROM table1;
    OPEN C1;
    CLOSE C1;
    :
    : -- more statements
  END INNER;
  :
  : -- more statements
```

Example: In the following example, DB2 checks for SQLSTATE 42704 only for statements inside the A compound statement.

```
CREATE PROCEDURE EXIT_TEST ()
  LANGUAGE SQL
  BEGIN
    DECLARE OUT_BUFFER VARCHAR(80);
    DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

    A: BEGIN
      DECLARE EXIT HANDLER FOR NO_TABLE
      BEGIN
        SET OUT_BUFFER ='Table does not exist';
        END;

      -- Drop potentially nonexistent table:
      DROP TABLE JAVELIN;

      B: SET OUT_BUFFER ='Table dropped successfully';
      END;
      -- Copy OUT_BUFFER to some message table:
      C: INSERT INTO MESSAGES VALUES (OUT_BUFFER);
```

The following notes describe a possible flow for the preceding example:

1. A nested compound statement with label A confines the scope of the NO_TABLE exit handler to the statements that are specified in the A compound statement.
2. If the table JAVELIN does not exist, the DROP statement raises the NO_TABLE condition.
3. The exit handler for NO_TABLE is activated.
4. The variable OUT_BUFFER is set to the string 'Table does not exist.'
5. Execution continues with the INSERT statement. No more statements in the A compound statement are processed.

Example:

The following example illustrates the scope of different condition handlers.

```
CREATE PROCEDURE ERROR_HANDLERS(IN PARAM INTEGER)
  LANGUAGE SQL
  OUTER: BEGIN
    DECLARE I INTEGER;
    DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

    DECLARE EXIT HANDLER FOR
      SQLSTATE VALUE '38H02',
      SQLSTATE VALUE '38H04',
      SQLSTATE VALUE '38HI4',
      SQLSTATE VALUE '38H06'
    OUTER_HANDLER: BEGIN
      DECLARE TEXT VARCHAR(70);
      SET TEXT = SQLSTATE ||
        ' RECEIVED AND MANAGED BY OUTER ERROR HANDLER' ;
      RESIGNAL SQLSTATE VALUE '38HE0'
        SET MESSAGE_TEXT = TEXT;
      END OUTER_HANDLER;

    INNER: BEGIN
      DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H03'
        RESIGNAL SQLSTATE VALUE '38HI3'
        SET MESSAGE_TEXT = '38H03 MANAGED BY INNER ERROR HANDLER';

      DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H04'
        RESIGNAL SQLSTATE VALUE '38HI4'
```

```

        SET MESSAGE_TEXT = '38H04 MANAGED BY INNER ERROR HANDLER';

DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H05'
RESIGNAL SQLSTATE VALUE '38HI5'
        SET MESSAGE_TEXT = '38H05 MANAGED BY INNER ERROR HANDLER';

CASE PARAM
  WHEN 1 THEN                                -- (1)
    SIGNAL SQLSTATE VALUE '38H01'
    SET MESSAGE_TEXT =
      'EXAMPLE 1: ERROR SIGNED FROM INNER COMPOUND STMT';

  WHEN 2 THEN                                -- (2)
    SIGNAL SQLSTATE VALUE '38H02'
    SET MESSAGE_TEXT =
      'EXAMPLE 2: ERROR SIGNED FROM INNER COMPOUND STMT';

  WHEN 3 THEN                                -- (3)
    SIGNAL SQLSTATE VALUE '38H03'
    SET MESSAGE_TEXT =
      'EXAMPLE 3: ERROR SIGNED FROM INNER COMPOUND STMT';

  WHEN 4 THEN                                -- (4)
    SIGNAL SQLSTATE VALUE '38H04'
    SET MESSAGE_TEXT =
      'EXAMPLE 4: ERROR SIGNED FROM INNER COMPOUND STMT';

  ELSE
    SET I = 1; /*Do not do anything */
END CASE;
END INNER;

CASE PARAM
  WHEN 5 THEN                                -- (5)
    SIGNAL SQLSTATE VALUE '38H05'
    SET MESSAGE_TEXT =
      'EXAMPLE 5: ERROR SIGNED FROM OUTER COMPOUND STMT';
  WHEN 6 THEN                                -- (6)
    SIGNAL SQLSTATE VALUE '38H06'
    SET MESSAGE_TEXT =
      'EXAMPLE 6: ERROR SIGNED FROM OUTER COMPOUND STMT';
  ELSE                                        -- (7)
    SET I = 1; /*Do not do anything */
END CASE;
END OUTER

```

The following table summarizes the behavior of the preceding example:

Input value for PARAM	Expected behavior
1	SQLSTATE 38H01 is signaled from the INNER compound statement. Because no appropriate handler exists, the procedure terminates and returns the unhandled exception condition, 38H01 with SQLCODE -438, to the calling application.
2	SQLSTATE 38H02 is signaled from the INNER compound statement. The condition handler in the OUTER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HE0, is issued from within the body of the condition handler. This exception causes control to be returned to the end of the OUTER compound statement with exception condition 38HE0 and SQLCODE -438. The procedure terminates and returns the unhandled condition to the calling application.

Input value for PARM	Expected behavior
3	SQLSTATE 38H03 is signaled from the INNER compound statement. A condition handler within the INNER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HI3, is issued from within the body of the condition handler. Because no appropriate handler exists, the procedure terminates and returns the unhandled exception condition, 38HI3 with SQLCODE -438, to the calling application.
4	SQLSTATE 38H04 is signaled from the INNER compound statement. A condition handler within the INNER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HI4, is issued from within the body of the condition handler. A condition handler in the OUTER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HE0, is issued from within the body of the condition handler. This exception causes control to be returned to the end of the OUTER compound statement with exception condition 38HE0 and SQLCODE -438. The procedure terminates and returns the unhandled condition to the calling application.
5	SQLSTATE 38H05 is signaled from the OUTER compound statement. Because no appropriate handler exists, the procedure terminates and returns the unhandled exception condition, 38H05 with SQLCODE -438, to the calling application.
6	SQLSTATE 38H06 is signaled from the OUTER compound statement. A condition handler in the OUTER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HE0, is issued from within the body of the condition handler. This exception causes control to be returned to the end of the OUTER compound statement with exception condition 38HE0 and SQLCODE -438. The procedure terminates and returns the unhandled condition to the calling application.
7	The ELSE clause of the CASE statement executes and processes the SET statement. A successful completion code is returned to the calling application.

Example: In the following example SQL procedure, the condition handler for exception1 is not within the scope of the condition handler for exception0. If exception condition exception1 is raised in the body of the condition handler for exception0, no appropriate handler exists, and the procedure terminates with an unhandled exception.

```
CREATE PROCEDURE divide ( .....)
LANGUAGE SQL CONTAINS SQL
BEGIN
    DECLARE dn_too_long CHAR(5) DEFAULT 'abcde';

    -- Declare condition names -----
    DECLARE exception0 CONDITION FOR SQLSTATE '22001';
    DECLARE exception1 CONDITION FOR SQLSTATE 'xxxxx';

    -- Declare cursors -----
    DECLARE cursor1 CURSOR WITH RETURN FOR
        SELECT * FROM dept;

    -- Declare handlers -----
    DECLARE CONTINUE HANDLER FOR exception0
    BEGIN
        some SQL statement that causes an error 'xxxxx'
    END

    DECLARE CONTINUE HANDLER FOR exception1
    BEGIN
        ...
    END
```

```

-- Mainline of procedure -----
INSERT INTO DEPT (DEPTNO) VALUES (dn_too_long);
-- Assume that this statement results in SQLSTATE '22001'

OPEN CURSOR1;
END

```

Retrieving diagnostic information by using GET DIAGNOSTICS in a handler:

Handlers specify the action that an SQL procedure takes when a particular error or condition occurs. In some cases, you want to retrieve additional diagnostic information about the error or warning condition.

You can include a GET DIAGNOSTICS statement in a handler to retrieve error or warning information. If you include GET DIAGNOSTICS, it must be the first statement that is specified in the handler.

Example: Using GET DIAGNOSTICS to retrieve message text: Suppose that you create an SQL procedure, named `divide1`, that computes the result of the division of two integers. You include GET DIAGNOSTICS to return the text of the division error message as an output parameter:

```

CREATE PROCEDURE divide1
  (IN numerator INTEGER, IN denominator INTEGER,
  OUT divide_result INTEGER, OUT divide_error VARCHAR(1000))
LANGUAGE SQL
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    GET DIAGNOSTICS CONDITION 1 divide_error = MESSAGE_TEXT;
  SET divide_result = numerator / denominator;
END

```

Ignoring a condition in an SQL procedure:

You can specify that you want to ignore errors or warnings within a particular scope of statements in an SQL procedure. However, do so with caution.

To ignore a condition in an SQL procedure:

Declare a condition handler that contains an empty compound statement.

The following example shows a condition handler that is declared as a way of ignoring a condition. Assume that your SQL procedure inserts rows into a table that has a unique column. If the value to be inserted for that column already exists in the table, the row is not inserted. However, in this case, you do not want DB2 to notify the application about this condition, which is indicated by SQLSTATE 23505.

```

DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
BEGIN -- ignore error for duplicate value
END;

```

Related concepts:

“Handlers in an SQL procedure” on page 578

Related reference:

 [SQLSTATE values and common error codes \(DB2 Codes\)](#)

Raising a condition within an SQL procedure by using the SIGNAL or RESIGNAL statements

Within an SQL procedure, you can force a particular condition to occur with a specific SQLSTATE and message text.

You can use either a SIGNAL or RESIGNAL statement to raise a condition with a specific SQLSTATE and message text within an SQL procedure. The SIGNAL and RESIGNAL statements differ in the following ways:

- You can use the SIGNAL statement anywhere within an SQL procedure. You must specify the SQLSTATE value. In addition, you can use the SIGNAL statement in a trigger body. For information about using the SIGNAL statement in a trigger, see “Creating triggers” on page 493.
- You can use the RESIGNAL statement only within a handler of an SQL procedure. If you do not specify the SQLSTATE value, DB2 uses the same SQLSTATE value that activated the handler.

You can use any valid SQLSTATE value in a SIGNAL or RESIGNAL statement, except an SQLSTATE class with '00' as the first two characters.

The following table summarizes the differences between issuing a RESIGNAL or SIGNAL statement within the body of a condition handler. For each row in the table, assume that the diagnostics area contains the following information when the RESIGNAL or SIGNAL statement is issued:

```
RETURNED_SQLSTATE  xxxxx
MESSAGE_TEXT 'this is my message'
```

Table 93. Example RESIGNAL and SIGNAL statements

Specify a new condition?	Specify message text?	Example RESIGNAL statement...	Example SIGNAL statement...	Result
No	No	RESIGNAL 1	Not possible	RETURNED_SQLSTATE xxxxx MESSAGE_TEXT 'this is my message'
Yes	No	RESIGNAL '98765' 2	SIGNAL '98765'	RETURNED_SQLSTATE 98765 MESSAGE_TEXT 'APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT: this is my message'
No	Yes	Not possible	Not possible	NA
Yes	Yes	RESIGNAL '98765' SET MESSAGE_TEXT = 'xyz' 3	SIGNAL '98765' SET MESSAGE_TEXT = 'xyz' 3	RETURNED_SQLSTATE 98765 MESSAGE_TEXT 'APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT: xyz'

Note:

1. This statement raises the current condition with the existing SQLSTATE, SQLCODE, message text, and tokens.
2. This statement raises a new condition (SQLSTATE '98765'). Existing message text and tokens are reset. The SQLCODE is set to -438 for an error or 438 for a warning.

3. This statement raises a new condition (SQLSTATE '98765') with new message text ('xyz'). The SQLCODE is set to -438 for an error or 438 for a warning.

Example of the SIGNAL statement in an SQL procedure:

You can use the SIGNAL statement anywhere within an SQL procedure to raise a particular condition.

The following example uses an ORDERS table and a CUSTOMERS table that are defined in the following way:

```
CREATE TABLE ORDERS
  (ORDERNO    INTEGER NOT NULL,
   PARTNO     INTEGER NOT NULL,
   ORDER_DATE DATE DEFAULT,
   CUSTNO     INTEGER NOT NULL,
   QUANTITY   SMALLINT NOT NULL,
   CONSTRAINT REF_CUSTNO FOREIGN KEY (CUSTNO)
     REFERENCES CUSTOMERS (CUSTNO) ON DELETE RESTRICT,
   PRIMARY KEY (ORDERNO,PARTNO));

CREATE TABLE CUSTOMERS
  (CUSTNO     INTEGER NOT NULL,
   CUSTNAME   VARCHAR(30),
   CUSTADDR   VARCHAR(80),
   PRIMARY KEY (CUSTNO));
```

Example: Using SIGNAL to set message text

Suppose that you have 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 has a foreign key to the CUSTOMERS table, which requires that the CUSTNO exist in the CUSTOMERS table before an order can be inserted:

```
CREATE PROCEDURE submit_order
  (IN ONUM INTEGER, IN PNUM INTEGER,
   IN CNUM 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, PARTNO, CUSTNO, QUANTITY)
      VALUES (ONUM, PNUM, CNUM, QNUM);
  END
```

In this example, the SIGNAL statement is in the handler. However, you can use the SIGNAL statement to invoke a handler when a condition occurs that will result in an error.

Related concepts:

“Example of the RESIGNAL statement in a handler”

Example of the RESIGNAL statement in a handler:

You can use the RESIGNAL statement in an SQL procedure to assign a different value to the condition that activated the handler. T

Example: Using RESIGNAL to set an SQLSTATE valu

Suppose that you create an SQL procedure, named divide2, that computes the result of the division of two integers. You include SIGNAL to invoke the handler

with an overflow condition that is caused by a zero divisor, and you include RESIGNAL to set a different SQLSTATE value for that overflow condition:

```
CREATE PROCEDURE divide2
  (IN numerator INTEGER, IN denominator INTEGER,
  OUT divide_result INTEGER)
LANGUAGE 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
```

Example: RESIGNAL in a nested compound statement

If the following SQL procedure is invoked with argument values 1, 0, and 0, the procedure returns a value of 2 for RC and sets the oparm1 parameter to 650.

```
CREATE PROCEDURE resig4
  (IN iparm1 INTEGER, INOUT oparm1 INTEGER, INOUT rc INTEGER)
LANGUAGE SQL
A1: BEGIN
  DECLARE c1 INT DEFAULT 1;
  DECLARE CONTINUE HANDLER FOR SQLSTATE VALUE '01ABX'
  BEGIN
    ... some other statements
    SET RC = 3; 6
  END;

  A2: SET oparm1 = 5; 1

  A3: BEGIN
    DECLARE c1 INT DEFAULT 1;
    DECLARE CONTINUE HANDLER
      FOR SQLSTATE VALUE '01ABC'
    BEGIN
      SET RC = 1; 4
      RESIGNAL SQLSTATE VALUE '01ABX' 5
      SET MESSAGE_TEXT = 'get out of here';
      SET RC = 2; 7
    END;

    A7: SET oparm1 = oparm1 + 110; 2
    SIGNAL SQLSTATE VALUE '01ABC' 3
    SET MESSAGE_TEXT = 'yikes';
    SET oparm1 = oparm1 + 215; 8
  END;
  SET oparm1 = oparm1 + 320; 9
END
```

The following notes refer to the preceding example:

1. oparm1 is initially set to 5.
2. oparm1 is incremented by 110. The value of oparm1 is now 115.
3. The SIGNAL statement causes the condition handler that is contained in the A3 compound statement to be activated.
4. In this condition handler, RC is set to 1.
5. The RESIGNAL statement changes the SQLSTATE to 01ABX. This value causes the continue handler in the A1 compound statement to be activated.

6. RC is set to 3 in this condition handler. Because this condition handler is a continue handler, when the handler action completes, control returns to the SET statement after the RESIGNAL statement.
7. RC is set to 2 in this condition handler. Because this condition handler is a continue handler, control returns to the SET statement that follows the SIGNAL statement that caused the condition handler to be activated.
8. oparm1 is incremented by 215. The value of oparm is now 330.
9. oparm1 is incremented by 320. The value of oparm is now 650.

How SIGNAL and RESIGNAL statements affect the diagnostics area:

When you issue a SIGNAL statement, a new logical diagnostics area is created. When you issue a RESIGNAL statement, the current diagnostics area is updated.

When you issue a SIGNAL statement, a new diagnostics area is logically created. In that diagnostics area, RETURNED_SQLSTATE is set to the SQLSTATE or condition name specified. If you specified message text as part of the SIGNAL statement, MESSAGE_TEXT in the diagnostics area is also set to the specified value.

When you issue a RESIGNAL statement with a SQLSTATE value, condition name, or message text, the current diagnostics area is updated with the specified information.

Making copies of a package for a native SQL procedure

When you create a native SQL procedure, a package is implicitly bound with the options that you specified on the CREATE PROCEDURE statement. If the native SQL procedure performs certain actions, you need to explicitly make copies of that package.

If the native SQL procedure performs one or more of the following actions, you need to create copies of the package for that procedure:

- Uses a CONNECT statement to connect to a database server.
- Refers to a table with a three part name that includes a location other than the current server or refers to an alias that resolves to such a name.
- Sets the CURRENT PACKAGESET special register to control which package is invoked for that version of the procedure.
- Sets the CURRENT PACKAGE PATH special register to control which package is invoked for that version of the procedure.

The package for a version of a procedure has the following name: *location.collection-id.package-id.version-id* where these variables have the following values:

location

Value of the CURRENT SERVER special register

collection-id

Schema qualifier of the procedure

package-id

Procedure name

version-id

Version identifier

To make copies of a package for a native SQL procedure, specify the BIND PACKAGE command with the COPY option. For copies that are created on the current server, specify a different schema qualifier, which is the collection ID. For the first copy that is created on a remote server, you can specify the same schema qualifier. For other copies on that remote server, specify a different schema qualifier.

If you later change the native SQL procedure, you might need to explicitly rebind any local or remote copies of the package that exist for that version of the procedure.

Example: Because the following native SQL procedure contains a CONNECT statement, you must create a copy of the package at the target server, which in this case is at location SAN_JOSE. The subsequent BIND command creates a copy of the package for version ABC of the procedure TEST.MYPROC. This package is created at location SAN_JOSE and is used by DB2 when this procedure is executed.

```
CREATE PROCEDURE TEST.MYPROC VERSION ABC LANGUAGE SQL ...
BEGIN
    ...
    CONNECT TO SAN_JOSE
    ...
END

BIND PACKAGE (SAN_JOSE.TEST) COPY(TEST.MYPROC) COPYVER(ABC) ACTION(ADD)
```

Example: The following native SQL procedure sets the CURRENT PACKAGESET special register to ensure that DB2 uses the package with the collection ID COLL2 for this version of the procedure. Consequently, you must create such a package. The subsequent BIND command creates this package with collection ID COLL2. This package is a copy of the package for version ABC of the procedure TEST.MYPROC. DB2 uses this package to process the SQL statements in this procedure.

```
CREATE PROCEDURE TEST.MYPROC VERSION ABC LANGUAGE SQL ...
BEGIN
    ...
    SET CURRENT PACKAGESET = 'COLL2'
    ...
END


BIND PACKAGE(COLL2) COPY(TEST.MYPROC) COPYVER(ABC)
    ACTION(ADD) QUALIFIER(XYZ)
```

Related tasks:

“Regenerating an existing version of a native SQL procedure” on page 599

“Replacing copies of a package for a version of a native SQL procedure”

Related reference:

 ALTER PROCEDURE (SQL - native) (DB2 SQL)

Replacing copies of a package for a version of a native SQL procedure:

When you change a version of a native SQL procedure and the ALTER PROCEDURE REPLACE statement contains certain options, you must replace any local or remote copies of the package that exist for that version of the procedure.

If you specify any of the following ALTER PROCEDURE options, you must replace copies of the package:

- REPLACE VERSION
- REGENERATE
- DISABLE DEBUG MODE
- QUALIFIER
- PACKAGE OWNER
- DEFER PREPARE
- NODEFER PREPARE
- CURRENT DATA
- DEGREE
- DYNAMICRULES
- APPLICATION ENCODING SCHEME
- WITH EXPLAIN
- WITHOUT EXPLAIN
- WITH IMMEDIATE WRITE
- WITHOUT IMMEDIATE WRITE
- ISOLATION LEVEL
- WITH KEEP DYNAMIC
- WITHOUT KEEP DYNAMIC
- OPTHINT
- SQL PATH
- RELEASE AT COMMIT
- RELEASE AT DEALLOCATE
- REOPT
- VALIDATE RUN
- VALIDATE BIND
- ROUNDING
- DATE FORMAT
- DECIMAL
- FOR UPDATE CLAUSE OPTIONAL
- FOR UPDATE CLAUSE REQUIRED
- TIME FORMAT

To replace copies of a package for a version of a native SQL procedure, specify the BIND COPY ACTION(REPLACE) command with the appropriate package name and version ID.

Creating a new version of a native SQL procedure

A new version of a native SQL procedure can have different parameter names, procedure options, or procedure body.

All versions of a procedure must have the same procedure signature. Therefore, each version of the procedure must have the same of the following items:

- Schema name
- Procedure name
- Number of parameters
- Data types for corresponding parameters

To create a new version of a procedure:

Issue the ALTER PROCEDURE statement with the following items:

- The name of the native SQL procedure for which you want to create a new version.
- The ADD VERSION clause with a name for the new version.
- The parameter list of the procedure that you want to alter. This parameter list must be the same as the original procedure.
- Any procedure options. These options can be different than the options for other versions of this procedure. If you do not specify a value for a particular option, the default value is used, regardless of the value that is used by the current active version of this procedure.
- A procedure body. This body can be different than the procedure body for other versions of this procedure.

For example, the following CREATE PROCEDURE statement defines a new native SQL procedure called UPDATE_BALANCE. The version of the procedure is V1, and it is the active version.

```
CREATE PROCEDURE
UPDATE_BALANCE
(IN CUSTOMER_NO INTEGER,
 IN AMOUNT DECIMAL(9,2))
VERSION V1
LANGUAGE SQL
READS SQL DATA
BEGIN
DECLARE CUSTOMER_NAME CHAR(20);
SELECT CUSTNAME
INTO CUSTOMER_NAME
FROM ACCOUNTS
WHERE CUSTNO = CUSTOMER_NO;
END
```

The following ALTER PROCEDURE statement creates a new version of the UPDATE_BALANCE procedure. The version name of the new version is V2. This new version has a different procedure body.

```
ALTER PROCEDURE
UPDATE_BALANCE
ADD VERSION V2
(IN CUSTOMER_NO INTEGER,
 IN AMOUNT DECIMAL (9,2) )
MODIFIES SQL DATA
BEGIN
UPDATE ACCOUNTS
SET BAL = BAL + AMOUNT
WHERE CUSTNO = CUSTOMER_NO;
END
```

After you create a new version, if you want that version to be invoked by all subsequent calls to this procedure, you need to make that version the active version.

Multiple versions of native SQL procedures:

You can define multiple versions of a native SQL procedure. DB2 maintains this version information for you.

One or more versions of a procedure can exist at any point in time at the current server, but only one version of a procedure is considered the active version. When you first create a procedure, that initial version is considered the active version of the procedure.

Using multiple versions of a native SQL procedure has the following advantages:

- You can keep the existing version of a procedure active while you create another version. When the other version is ready, you can make it the active one.
- When you make another version of a procedure active, you do not need to change any existing calls to that procedure.
- You can easily switch back to a previous version of a procedure if the version that you switched to does not work as planned.
- You can drop an unneeded version of a procedure.

A new version of a native SQL procedure can have different values for the following items:

- Parameter names
- Procedure options
- Procedure body

Restrictions:

- A new version of a native SQL procedure cannot have different values for the following items:
 - Number of parameters
 - Parameter data types
 - Parameter attributes for character data
 - Parameter CCSIDs
 - Whether a parameter is an input or output parameter, as defined by the IN, OUT, and INOUT options

If you need to specify different values for any of the preceding items, create a new native SQL procedure, instead of a new version.

Deploying a native SQL procedure to another DB2 for z/OS server

When deploying a native SQL procedure to another DB2 for z/OS server, you can change the bind options to better match the deploying environment. The procedure logic remains the same. This deployment process is useful when you want to move a procedure from a test system to a production system.

Requirements:

- The remote server must be properly defined in the communications database of the DB2 subsystem from which you deploy the native SQL procedure.
- The target DB2 subsystem must be operating at a PTF level that is compatible with the PTF level of the local DB2 subsystem.

To deploy a native SQL procedure to another DB2 for z/OS server:

Issue the BIND PACKAGE command with the following options:

DEPLOY

Specify the name of the procedure whose logic you want to use on the target server.

Tip: When specifying the parameters for the DEPLOY option, consider the following naming rules for native SQL procedures:

- The collection ID is the same as the schema name in the original CREATE PROCEDURE statement.
- The package ID is the same as the procedure name in the original CREATE PROCEDURE statement.

COPYVER

Specify the version of the procedure whose logic you want to use on the target server.

ACTION(ADD) or ACTION(REPLACE)

Specify whether you want DB2 to create a new version of the native SQL procedure and its associated package or to replace the specified version.

Optionally, you can also specify the bind options QUALIFIER or OWNER if want to change them.

Example of deploying the same version of a procedure at another location: The following BIND command creates a native SQL procedure with the name PRODUCTION.MYPROC at the CHICAGO location. This procedure is created from the procedure TEST.MYPROC at the current site. Both native SQL procedures have the same content and version, ABC. However, the package for the procedure CHICAGO.PRODUCTION.MYPROC has XYZ as its qualifier.

```
CREATE PROCEDURE TEST.MYPROC VERSION ABC LANGUAGE SQL ...
BEGIN
...
END

BIND PACKAGE(CHICAGO.PRODUCTION) DEPLOY(TEST.MYPROC) COPYVER(ABC)
ACTION(ADD) QUALIFIER(XYZ)
```


Example of replacing a version of a procedure at another location: The following BIND command replaces version ABC of the procedure PRODUCTION.MYPROC at the CHICAGO location with version ABC of the procedure TEST.MYPROC at the current site.

```
BIND PACKAGE(CHICAGO.PRODUCTION) DEPLOY(TEST.MYPROC) COPYVER(ABC)
ACTION(REPLACE) REPLVER(ABC)
```


Related concepts:

 Communications database for the server (Managing Security)

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

 BIND PACKAGE (DSN) (DB2 Commands)

 Scenario for deployment (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

Migrating an external SQL procedure to a native SQL procedure

Native SQL procedures typically perform better, have more functionality, and are easier to maintain than external SQL procedures.

If you created the external SQL procedure in a previous release of DB2, consider the release incompatibilities for applications that use stored procedures. For information about the release incompatibilities, see “Application and SQL release

incompatibilities for migration from Version 9.1” on page 32 or “Application and SQL release incompatibilities for migration from Version 8” on page 1. Examine your external SQL procedure source code, and make any necessary adjustments.

To migrate an external SQL procedure to a native SQL procedure:

1. Find and save the existing CREATE PROCEDURE and GRANT EXECUTE statements for the existing external SQL procedure.
2. Drop the existing external SQL procedure by using the DROP PROCEDURE statement.
3. Re-create the procedure as a native SQL procedure by using the same CREATE PROCEDURE statement that you used to originally create the procedure, with both of the following changes:
 - If the procedure was defined with the options FENCED or EXTERNAL, remove these keywords.
 - Either remove the WLM ENVIRONMENT keyword, or add the FOR DEBUG MODE clause.
 - If the procedure body contains statements with unqualified names that could refer to either a column or an SQL variable or parameter, qualify these names. Otherwise, you might need to change the statement.

DB2 resolves these names differently depending on whether the procedure is an external SQL procedure or a native SQL procedure. For external SQL procedures, DB2 first treats the name as a variable or parameter if one exists with that name. For native SQL procedures, DB2 first treats the name as a column if a column exists with that name. For example, consider the following statement:

```
CREATE PROCEDURE P1 (INOUT C1 INT) ... SELECT C1 INTO xx FROM T1
```

In the preceding example, if P1 is an external SQL procedure, C1 is a parameter. For native SQL procedures, C1 is a column in table T1. If such a column does not exist, C1 is a parameter.

4. Issue the same GRANT EXECUTE statements that you used to originally grant privileges for this stored procedure.
5. Increase the value of the TIME parameter on the job statement for applications that call stored procedures.

Important: This change is necessary because time for SQL external stored procedures is charged to the WLM address space, while time for native SQL stored procedures is charged to the address space of the task.

6. Test your new native SQL procedure.

Related tasks:

 [Implementing DB2 stored procedures \(DB2 Administration Guide\)](#)

Related reference:

 [CREATE PROCEDURE \(SQL - external\) \(DB2 SQL\)](#)

 [CREATE PROCEDURE \(SQL - native\) \(DB2 SQL\)](#)

 [DROP \(DB2 SQL\)](#)

 [GRANT \(function or procedure privileges\) \(DB2 SQL\)](#)

Using the DB2 precompiler to assist you in converting an external SQL procedure to a native SQL procedure

The DB2 precompiler can be useful when considering any conversion of an external SQL procedure to a native SQL procedure.

Use the DB2 precompiler to inspect the SQL procedure source from a native SQL PL perspective. A listing is produced that helps to isolate problems and incompatibilities between external and native SQL procedure coding. Source changes can then be made before making any changes in DB2.

To inspect the quality of native SQL PL source coding using the DB2 precompiler:

1. Copy the original SQL PL source code to a FB80 data set. Reformat the source as needed to fit within the precompiler margins.
2. Precompile the SQL PL source by executing program DSNHPSM with the HOST(SQLPL) option.
3. Inspect the produced listing (SYSPRINT). Pay attention to error and warning messages.
4. Modify the SQL PL source to address coding problems that are identified by messages in the listing.
5. Repeat steps 1 through 4 until all error and warning messages are resolved. Address informational messages as needed.
6. Copy the modified SQL PL source file back to its original source format, reformatting as needed.

Sample JCL DSNTEJ67 demonstrates this process for an external SQL procedure that was produced using the DB2 SQL procedure processor DSNTPSMP.

Related reference:

“Sample programs to help you prepare and run external SQL procedures” on page 614

Changing an existing version of a native SQL procedure

You can change an option or the procedure body for a particular version of a native SQL procedure. If you want to keep a copy of that stored procedure, consider creating a new version instead of changing the existing version.

To change an existing version of a native SQL procedure:

Issue the ALTER PROCEDURE statement with the REPLACE VERSION clause. Any option that you do not explicitly specify inherits the system default values. This inheritance occurs even if those options were explicitly specified for a prior version by using a CREATE PROCEDURE statement, ALTER PROCEDURE statement, or REBIND command.




The following ALTER PROCEDURE statement updates version V2 of the UPDATE_BALANCE procedure.

```
ALTER PROCEDURE
TEST.UPDATE_BALANCE
REPLACE VERSION V2
(IN CUSTOMER_NO INTEGER,
IN AMOUNT DECIMAL(9,2))
MODIFIES SQL DATA
ASUTIME LIMIT 100
BEGIN
UPDATE ACCOUNTS
SET BAL = BAL + AMOUNT
WHERE CUSTNO = CUSTOMER_NO
AND CUSTSTAT = 'A';
END
```

Related tasks:

“Creating a new version of a native SQL procedure” on page 593

Related reference:

-  [REBIND PACKAGE \(DSN\) \(DB2 Commands\)](#)
-  [ALTER PROCEDURE \(SQL - native\) \(DB2 SQL\)](#)
-  [CREATE PROCEDURE \(SQL - native\) \(DB2 SQL\)](#)

Regenerating an existing version of a native SQL procedure

When you apply DB2 maintenance that changes how native SQL procedures are generated, you need to regenerate any affected procedures. When you regenerate a version of a native SQL procedure, DB2 rebinds the associated package for that version of the procedure.

ALTER PROCEDURE REGENERATE is different than the REBIND PACKAGE command. When you specify REBIND PACKAGE, DB2 rebinds only the non-control SQL statements. Use this command when you think rebinding will improve the access path. When you specify ALTER PROCEDURE REGENERATE, DB2 rebinds the SQL control statements as well as the non-control statements.

To regenerate an existing version of a native SQL procedure:

1. Issue the ALTER PROCEDURE statement with the REGENERATE clause and specify the version to be regenerated.
2. If copies of the package for the specified version of the procedure exist at remote sites, replace those packages. Issue the BIND PACKAGE command with the COPY option and appropriate location for each remote package.
3. If copies of the package for the specified version of the procedure exist locally with different schema names, replace those packages. Issue the BIND PACKAGE command with the COPY option and appropriate schema for each local package.

The following ALTER PROCEDURE statement regenerates the active version of the UPDATE_SALARY_1 procedure.

```
ALTER PROCEDURE UPDATE_SALARY_1
REGENERATE ACTIVE VERSION
```

Removing an existing version of a native SQL procedure

You can drop a particular version of a native SQL procedure without dropping the other versions of the procedure.

Before you remove an existing version of a native SQL procedure, ensure that the version is not active. If the version is the active version, designate a different active version before proceeding.

To remove an existing version of a native SQL procedure:

Issue the ALTER PROCEDURE statement with the DROP VERSION clause and the name of the version that you want to drop. If you instead want to drop all versions of the procedure, use the DROP statement.

Example of dropping a version that is not active: The following statement drops the OLD_PRODUCTION version of the P1 procedure.

```
ALTER PROCEDURE P1 DROP VERSION OLD_PRODUCTION
```

Example of dropping an active version: Assume that the OLD_PRODUCTION version of the P1 procedure is the active version. The following example first switches the active version to NEW_PRODUCTION and then drops the OLD_PRODUCTION version.

```
ALTER PROCEDURE P1 ACTIVATE VERSION NEW_PRODUCTION;  
ALTER PROCEDURE P1 DROP VERSION OLD_PRODUCTION;
```

Related tasks:

“Designating the active version of a native SQL procedure” on page 804

Creating an external SQL procedure

An *external SQL procedure* is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language. However, an external SQL procedure is created, implemented, and executed like other external stored procedures. All SQL procedures that were created prior to Version 9 are external SQL procedures.

Before you create an external SQL procedure, Configure DB2 for running stored procedures and user-defined functions during installation or Configure DB2 for running stored procedures and user-defined functions during migration (DB2 Installation Guide).

To create an external SQL procedure:

1. Use one of the following methods to create the external SQL procedure:
 - IBM Data Studio. See Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio).
 - JCL
 - The DB2 for z/OS SQL procedure processor (DSNTPSMP)

The preceding methods that you use to create an external SQL procedure perform the following actions:

- Convert the external SQL procedure source statements into a C language program by using the DB2 precompiler
- Create an executable load module and a DB2 package from the C language program.
- Define the external SQL procedure to DB2 by issuing a CREATE PROCEDURE statement either statically or dynamically.

Restriction: If you plan to use the DB2 stored procedure debugger or the Unified Debugger, do not use JCL. Use either IBM Data Studio or DSNTPSMP.

If you plan to use IBM Data Studio or DSNTPSMP, you must set up support for external SQL procedures.

2. Authorize the appropriate users to use the stored procedure by issuing the GRANT EXECUTE statement.

For examples of how to prepare and run external SQL procedures, see “Sample programs to help you prepare and run external SQL procedures” on page 614.

Related concepts:

“SQL procedures” on page 565

Related tasks:

 Implementing DB2 stored procedures (DB2 Administration Guide)

Related reference:

➡ CREATE PROCEDURE (SQL - external) (DB2 SQL)

➡ GRANT (function or procedure privileges) (DB2 SQL)

Creating an external SQL procedure by using DSNTPSMP

The SQL procedure processor, DSNTPSMP, is one of several methods that you can use to create and prepare an external SQL procedure. DSNTPSMP is a REXX stored procedure that you can invoke from your application program.

Set up support for external SQL procedures.

Also ensure that you have the required authorizations, as indicated in the following table, for invoking DSNTPSMP.

Table 94. Required authorizations for invoking DSNTPSMP

Required authorization	Associated syntax for the authorization
Procedure privilege to run application programs that invoke the stored procedure.	EXECUTE ON PROCEDURE SYSPROC.DSNTPSMP
Collection privilege to use BIND to create packages in the specified collection. You can use an asterisk (*) as the identifier for a collection.	CREATE ON COLLECTION <i>collection-id</i>
Package privilege to use BIND or REBIND to bind packages in the specified collection.	BIND ON PACKAGE <i>collection-id.*</i>
System privilege to use BIND with the ADD option to create packages and plans.	BINDADD
Schema privilege to create, alter, or drop stored procedures in the specified schema. The BUILDOWNER authorization ID must have the CREATEIN privilege on the schema. You can use an asterisk (*) as the identifier for a schema.	CREATEIN, ALTERIN, DROPIN ON SCHEMA <i>schema-name</i>
Table privileges to select or delete from, insert into, or update the specified catalog tables.	SELECT ON TABLE SYSIBM.SYSROUTINES SELECT ON TABLE SYSIBM.SYSPARMS SELECT, INSERT, UPDATE, DELETE ON TABLE SYSIBM.SYSROUTINES_SRC SELECT, INSERT, UPDATE, DELETE ON TABLE SYSIBM.SYSROUTINES_OPTS ALL ON TABLE SYSIBM.SYSPSMOUT
Any privileges that are required for the SQL statements and that are contained within the SQL procedure body. These privileges must be associated with the OWNER <i>authorization-id</i> that is specified in your bind options. The default owner is the user that is invoking DSNTPSMP.	Syntax varies depending on the SQL procedure body

To create an external SQL procedure by using DSNTPSMP:

1. Write an application program that calls DSNTPSMP. Include the following items in your program:

- A CLOB host variable that contains a CREATE PROCEDURE statement for the external SQL procedure. That statement should include the FENCED keyword or the EXTERNAL keyword, and the procedure body, which is written in SQL.

Alternatively, instead of defining a host variable for the CREATE PROCEDURE statement, you can store the statement in a data set member.

- An SQL CALL statement with the BUILD function. The CALL statement should use the proper syntax for invoking DSNTPSMP.

Pass the SQL procedure source to DSNTPSMP as one of the following input parameters:

SQL-procedure-source

Use this parameter if you defined a host variable in your application to contain the CREATE PROCEDURE statement.

source-data-set-name

Use this parameter if you stored the CREATE PROCEDURE statement in a data set.

- Based on the return value from the CALL statement, issue either an SQL COMMIT or a ROLLBACK statement. If the return value is 0 or 4, issue a COMMIT statement. Otherwise, issue a ROLLBACK statement.

You must process the result set before issuing the COMMIT or ROLLBACK statement.

A QUERYLEVEL request must be followed by the COMMIT statement.

2. Precompile, compile, and link-edit the application program.
3. Bind a package for the application program.
4. Run the application program.

Related concepts:

“SQL procedure body” on page 566

Related reference:

 CREATE PROCEDURE (SQL - external) (DB2 SQL)

DB2 for z/OS SQL procedure processor (DSNTPSMP):

The SQL procedure processor, DSNTPSMP, is a REXX stored procedure that you can use to prepare an external SQL procedure for execution.

You can also use DSNTPSMP to perform selected steps in the preparation process or delete an existing external SQL procedure. DSNTPSMP is the only preparation method for enabling external SQL procedures to be debugged with either the SQL Debugger or the Unified Debugger.

DSNTPSMP requires that your system EBCDIC CCSID also be compatible with the C compiler. Using an incompatible CCSID results in compile-time errors. Examples of incompatible CCSIDs include 290, 930, 1026, and 1155. If your system EBCDIC CCSID is not compatible, do not just change it. Contact IBM Software Support for help.

Sample startup procedure for a WLM address space for DSNTPSMP:

You must run DSNTPSMP in a WLM-established stored procedures address space. You should run only DSNTPSMP in that address space, and you must limit the address space to run only one task concurrently.

This example shows how to set up a WLM address space for DSNTPSMP.

Recommendation: Use the core WLM environment DSNWLM_REXX. Job DSNTIJMV creates an address space procedure called DSNWLMR for this environment.

The following example shows sample JCL for a startup procedure for the address space in which DSNTPSMP runs.

```

//DSNWLMR PROC DB2SSN=DSN,NUMTCB=1,APPLENV=DSNWLM_REXX           1
/*
//WLMTPSMP EXEC PGM=DSNX9WLM,TIME=1440,                          2
//          PARM='&DB2SSN,&NUMTCB,&APPLENV',
//          REGION=0M,DYNAMNBR=10
//STEPLIB DD DISP=SHR,DSN=DSN1010.SDSNEXIT                       3
//          DD DISP=SHR,DSN=DSN1010.SDSNLOAD
//          DD DISP=SHR,DSN=CBC.SCCNCMP
//          DD DISP=SHR,DSN=CEE.SCEERUN
//          DD DISP=SHR,DSN=DSN1010.DBRMLIB.DATA                 3
//SYSEXEC DD DISP=SHR,DSN=DSN1010.SDSNCLST                       4
//SYSTSPT DD SYSOUT=A
//CEEDUMP DD SYSOUT=A
//SYSABEND DD DUMMY
/*
//SQLDBRM DD DISP=SHR,DSN=DSN1010.DBRMLIB.DATA                 5
//SQLCSRC DD DISP=SHR,DSN=DSN1010.SRCLIB.DATA                   6
//SQLMOD DD DISP=SHR,DSN=DSN1010.RUNLIB.LOAD                    7
//SQLLIBC DD DISP=SHR,DSN=CEE.SCEEH.H                           8
//          DD DISP=SHR,DSN=CEE.SCEEH.SYS.H
//SQLLIBL DD DISP=SHR,DSN=CEE.SCEELKED                           9
//          DD DISP=SHR,DSN=DSN1010.SDSNLOAD
//SYSMSGSGS DD DISP=SHR,DSN=CEE.SCEEMSGP(EDCPMSGGE)            10
/*
/* DSNTPSMP Configuration File - CFGTPSMP (optional)            11
/*      A site-provided sequential data set or member, used to
/*      define customized operation of DSNTPSMP in this APPLNV
/*
/* CFGTPSMP DD DISP=SHR,DSN=
/*
//SQLSRC DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),                12
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLPRINT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLTERM DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLOUT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLCPRD DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//          DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLUT1 DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLUT2 DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLCIN DD UNIT=SYSALLDA,SPACE=(32000,(20,20))
//SQLLIN DD UNIT=SYSALLDA,SPACE=(3200,(30,30)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSMOD DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLDUMMY DD DUMMY

```

Notes:

- 1** APPLNV specifies the application environment in which DSNTPSMP runs. To ensure that DSNTPSMP always uses the correct data sets and parameters for preparing each external SQL procedure, you can set up


different application environments for preparing stored procedures with different program preparation requirements. For example, if all payroll applications use the same set of data sets during program preparation, you could set up an application environment called PAYROLL for preparing only payroll applications. The startup procedure for PAYROLL would point to the data sets that are used for payroll applications.

DB2SSN specifies the DB2 subsystem name.

NUMTCB specifies the number of programs that can run concurrently in the address space. You should always set NUMTCB to 1 to ensure that executions of DSNTPSMP occur serially.

- 2** WLMTPSMP specifies the address space in which DSNTPSMP runs.
DYNAMNBR reserves space for dynamic allocation of data sets during the SQL procedure preparation process.
- 3** STEPLIB specifies the DB2 load libraries, the z/OS C/C++ compiler library, and the Language Environment run time library that DSNTPSMP uses when it runs. At least one library must not be APF authorized.
- 4** SYSEXEC specifies the library that contains the REXX exec DSNTPSMP.
- 5** SQLDBRM is an output data set that specifies the library into which DSNTPSMP puts the DBRM that it generates when it precompiles your external SQL procedure.
- 6** SQLCSRC is an output data set that specifies the library into which DSNTPSMP puts the C source code that it generates from the external SQL procedure source code. This data set should have a logical record length of 80.
- 7** SQLLMOD is an output data set that specifies the library into which DSNTPSMP puts the load module that it generates when it compiles and link-edits your external SQL procedure.
- 8** SQLLIBC specifies the library that contains standard C header files. This library is used during compilation of the generated C program.
- 9** SQLLIBL specifies the following libraries, which DSNTPSMP uses when it link-edits the external SQL procedure:
 - Language Environment link-edit library
 - DB2 load library
- 10** SYSMSGs specifies the library that contains messages that are used by the C prelink-edit utility.
- 11** CFGTPSMP specifies an optional data set that you can use to customize DSNTPSMP, including specifying the compiler level. For details on all of the options that you can set in this file and how to set them, see the DSNTPSMP CLIST comments.
- 12** The DD statements that follow describe work file data sets that are used by DSNTPSMP.

Related tasks:

 Converting from the AMI-based MQ functions to the MQI-based MQ functions (DB2 Installation and Migration)

CALL statement syntax for invoking DSNTPSMP:

You can invoke the SQL procedure processor, DSNTPSMP, from an application program by using an SQL CALL statement. DSNTPSMP prepares an external SQL procedure.

The following diagrams show the syntax of invoking DSNTPSMP through the SQL CALL statement:

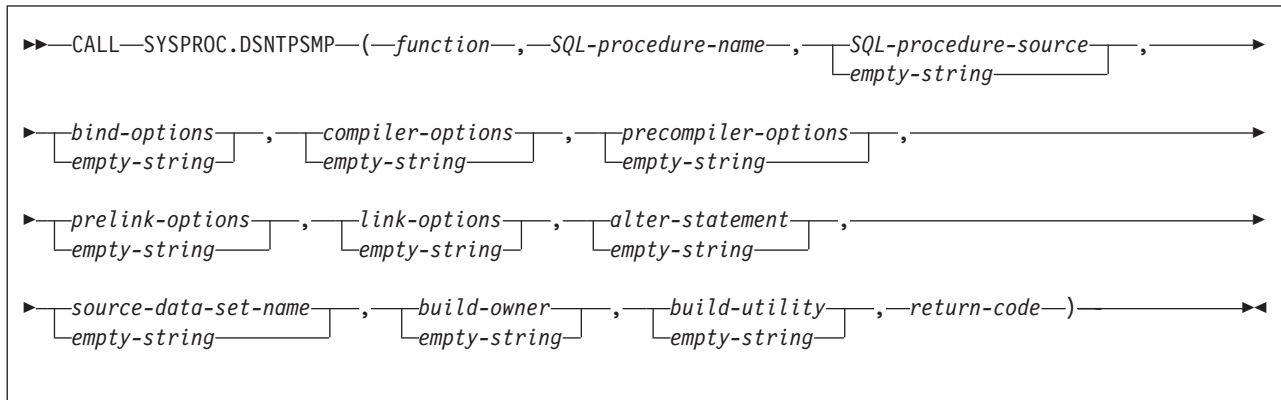


Figure 31. DSNTPSMP syntax

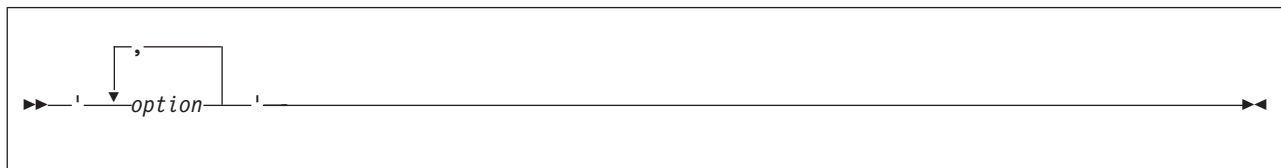


Figure 32. CALL DSNTPSMP bind-options, compiler-options, precompiler-options, prelink-options, link-options

Note: You must specify:

- The DSNTPSMP parameters in the order listed
- The empty string if an optional parameter is not required for the function
- The options in the order: bind, compiler, precompiler, prelink, and link

The DSNTPSMP parameters are:

function

A VARCHAR(20) input parameter that identifies the task that you want DSNTPSMP to perform. The tasks are:

BUILD

Creates the following objects for an external SQL procedure:

- A DBRM, in the data set that DD name SQLDBRM points to
- A load module, in the data set that DD name SQLLMOD points to
- The C language source code for the external SQL procedure, in the data set that DD name SQLCSRC points to
- The stored procedure package
- The stored procedure definition

The following input parameters are required for the BUILD function:

SQL-procedure name

SQL-procedure-source or *source-data-set-name*

If you choose the BUILD function, and an external SQL procedure with name *SQL-procedure-name* already exists, DSNTPSMP issues an error message and terminates.

BUILD_DEBUG

Creates the following objects for an external SQL procedure and includes the preparation necessary to debug the external SQL procedure with the SQL Debugger and the Unified Debugger:

- A DBRM, in the data set that DD name SQLDBRM points to
- A load module, in the data set that DD name SQLLMOD points to
- The C language source code for the external SQL procedure, in the data set that DD name SQLCSRC points to
- The stored procedure package
- The stored procedure definition

The following input parameters are required for the BUILD_DEBUG function:

SQL-procedure name
SQL-procedure-source or *source-data-set-name*

If you choose the BUILD_DEBUG function, and an external SQL procedure with name *SQL-procedure-name* already exists, DSNTPSMP issues an error message and terminates.

REBUILD

Replaces all objects that were created by the BUILD function for an external SQL procedure, if it exists, otherwise creates those objects.

The following input parameters are required for the REBUILD function:

SQL-procedure name
SQL-procedure-source or *source-data-set-name*

REBUILD_DEBUG

Replaces all objects that were created by the BUILD_DEBUG function for an external SQL procedure, if it exists, otherwise creates those objects, and includes the preparation necessary to debug the external SQL procedure with the SQL Debugger and the Unified Debugger.

The following input parameters are required for the REBUILD_DEBUG function:

SQL-procedure name
SQL-procedure-source or *source-data-set-name*

REBIND

Binds the external SQL procedure package for an existing external SQL procedure.

The following input parameter is required for the REBIND function:

SQL-procedure name

DESTROY

Deletes the following objects for an existing external SQL procedure:

- The DBRM, from the data set that DD name SQLDBRM points to
- The load module, from the data set that DD name SQLLMOD points to
- The C language source code for the external SQL procedure, from the data set that DD name SQLCSRC points to
- The stored procedure package
- The stored procedure definition

The following input parameter is required for the DESTROY function:
SQL-procedure name

ALTER

Updates the registration for an existing external SQL procedure.

The following input parameters are required for the ALTER function:

SQL-procedure name

alter-statement

ALTER_REBUILD

Updates an existing external SQL procedure.

The following input parameters are required for the ALTER_REBUILD function:

SQL-procedure name

SQL-procedure-source or *source-data-set-name*

ALTER_REBUILD_DEBUG

Updates an existing external SQL procedure, and includes the preparation necessary to debug the external SQL procedure with the SQL Debugger and the Unified Debugger.

The following input parameters are required for the ALTER_REBUILD_DEBUG function:

SQL-procedure name

SQL-procedure-source or *source-data-set-name*

ALTER_REBIND

Updates the registration and binds the SQL package for an existing external SQL procedure.

The following input parameters are required for the ALTER_REBIND function:

SQL-procedure name

alter-statement

QUERYLEVEL

Obtains the interface level of the build utility invoked. No other input is required.

SQL-procedure-name

A VARCHAR(261) input parameter that specifies the external SQL procedure name.

The name can be qualified or unqualified. The name must match the procedure name that is specified within the CREATE PROCEDURE statement that is provided in *SQL-procedure-source* or that is obtained from *source-data-set-name*. In addition, the name must match the procedure name that is specified within the ALTER PROCEDURE statement that is provided in *alter-statement*. Do not mix qualified and unqualified references.

SQL-procedure-source

A CLOB(2M) input parameter that contains the CREATE PROCEDURE statement for the external SQL procedure. If you specify an empty string for this parameter, you need to specify the name *source-data-set-name* of a data set that contains the external SQL procedure source code.

bind-options

A VARCHAR(1024) input parameter that contains the options that you want to specify for binding the external SQL procedure package. Do not specify the MEMBER or LIBRARY option for the DB2 BIND PACKAGE command.

compiler-options

A VARCHAR(255) input parameter that contains the options that you want to specify for compiling the C language program that DB2 generates for the external SQL procedure.

precompiler-options

A VARCHAR(255) input parameter that contains the options that you want to specify for precompiling the C language program that DB2 generates for the external SQL procedure. Do not specify the HOST option.

prelink-options

A VARCHAR(255) input parameter that contains the options that you want to specify for prelinking the C language program that DB2 generates for the external SQL procedure.

link-options

A VARCHAR(255) input parameter that contains the options that you want to specify for linking the C language program that DB2 generates for the external SQL procedure.

alter-statement

A VARCHAR(32672) input parameter that contains the SQL ALTER PROCEDURE statement to process with the ALTER or ALTER_REBIND function.

source-data-set-name

A VARCHAR(80) input parameter that contains the name of a z/OS sequential data set or partitioned data set member that contains the source code for the external SQL procedure. If you specify an empty string for this parameter, you need to provide the external SQL procedure source code in *SQL-procedure-source*.

build-owner

A VARCHAR(130) input parameter that contains the SQL identifier to serve as the build owner for newly created SQL stored procedures.

When this parameter is not specified, the value defaults to the value in the CURRENT SQLID special register when the build utility is invoked.

build-utility

A VARCHAR(255) input parameter that contains the name of the build utility that is invoked. The qualified form of the name is suggested, for example, SYSPROC.DSNTPSMP.

return-code

A VARCHAR(255) output parameter in which DB2 puts the return code from the DSNTPSMP invocation. The values are:

- 0** Successful invocation. The calling application can optionally retrieve the result set and then issue the required SQL COMMIT statement.
- 4** Successful invocation, but warnings occurred. The calling application should retrieve the warning messages in the result set and then issue the required SQL COMMIT statement.
- 8** Failed invocation. The calling application should retrieve the error messages in the result set and then issue the required SQL ROLLBACK statement.

99x

Where *x* is a digit between 0 and 9. Failed invocation with severe errors. The calling application should retrieve the error messages in the result set

and then issue the required SQL ROLLBACK statement. To view error messages that are not in the result set, see the job log of the address space for the DSNTPSMP execution.


- 999 Unknown severe internal error
- 998 APF environment setup error
- 997 DSNREXX setup error
- 996 Global temporary table setup error
- 995 Internal REXX programming error


1.2x

Where *x* is a digit between 0 and 9. Level of DSNTPSMP when request is QUERYLEVEL. The calling application can retrieve the result set for additional information about the release and service level and then issue the required SQL COMMIT statement.

Related reference:

“Descriptions of SQL processing options” on page 932

 BIND and REBIND options for packages and plans (DB2 Commands)

 Compiler Options (C/C++) (XL C/C++ User's Guide)

 Binder options reference (MVS Program Management: User's Guide and Reference)

Examples of invoking the SQL procedure processor (DSNTPSMP):

You can invoke the BUILD, DESTROY, REBUILD, and REBIND functions of DSNTPSMP.

DSNTPSMP BUILD function: Call DSNTPSMP to build an external SQL procedure. The information that DSNTPSMP needs is listed in the following table:

Table 95. The functions DSNTPSMP needs to BUILD an SQL procedure

Function	BUILD
External SQL procedure name	MYSHEMA.SQLPROC
Source location	String in CLOB host variable procsrc
Bind options	VALIDATE(BIND)
Compiler options	SOURCE, LIST, LONGNAME, RENT
Precompiler options	SOURCE, XREF, STDSQL(NO)
Prelink options	None specified
Link options	AMODE=31, RMODE=ANY, MAP, RENT
Build utility	SYSPROC.DSNTPSMP
Return value	String returned in varying-length host variable returnval

The CALL statement is:

```
EXEC SQL CALL SYSPROC.DSNTPSMP('BUILD', 'MYSHEMA.SQLPROC', :procsrc,
    'VALIDATE(BIND)',
    'SOURCE, LIST, LONGNAME, RENT',
    'SOURCE, XREF, STDSQL(NO)');
```

```

'',
'AMODE=31,RMODE=ANY,MAP,RENT',
'', '', '', 'SYSPROC.DSNTPSMP',
:returnval);

```

DSNTPSMP DESTROY function: Call DSNTPSMP to delete an external SQL procedure definition and the associated load module. The information that DSNTPMSP needs is listed in the following table:

Table 96. The functions DSNTPSMP needs to DESTROY an SQL procedure

Function	DESTROY
External SQL procedure name	MYSHEMA.OLDPROC
Return value	String returned in varying-length host variable returnval

The CALL statement is:

```

EXEC SQL CALL SYSPROC.DSNTPSMP('DESTROY','MYSHEMA.OLDPROC','',
'', '', '', '', '',
'', '', '', '',
:returnval);

```

DSNTPSMP REBUILD function: Call DSNTPSMP to re-create an existing external SQL procedure. The information that DSNTPMSP needs is listed in the following table:

Table 97. The functions DSNTPSMP needs to REBUILD an SQL procedure

Function	REBUILD
External SQL procedure name	MYSHEMA.SQLPROC
Bind options	VALIDATE(BIND)
Compiler options	SOURCE, LIST, LONGNAME, RENT
Precompiler options	SOURCE, XREF, STDSQL(NO)
Prelink options	None specified
Link options	AMODE=31, RMODE=ANY, MAP, RENT
Source data set name	Member PROCSRC of partitioned data set DSNA10.SDSNSAMP
Return value	String returned in varying-length host variable returnval

The CALL statement is:

```

EXEC SQL CALL SYSPROC.DSNTPSMP('REBUILD','MYSHEMA.SQLPROC','',
'VALIDATE(BIND)',
'SOURCE,LIST,LONGNAME,RENT',
'SOURCE,XREF,STDSQL(NO)',
'',
'AMODE=31,RMODE=ANY,MAP,RENT',
'', 'DSNA10.SDSNSAMP(PROCSRC)', '', '',
:returnval);

```

If you want to re-create an existing external SQL procedure for debugging with the SQL Debugger and the Unified Debugger, use the following CALL statement, which includes the REBUILD_DEBUG function:

```
EXEC SQL CALL SYSPROC.DSNTPSMP('REBUILD_DEBUG','MYSCHEMA.SQLPROC','',
'VALIDATE(BIND)',
'SOURCE,LIST,LONGNAME,RENT',
'SOURCE,XREF,STDSQL(NO)',
'',
'AMODE=31,RMODE=ANY,MAP,RENT',
'', 'DSNA10.SDSNSAMP(PROCSRC)', '', '',
:returnval);
```

DSNTPSMP REBIND function: Call DSNTPSMP to rebind the package for an existing external SQL procedure. The information that DSNTPSMP needs is listed in the following table:

Table 98. The functions DSNTPSMP needs to REBIND an SQL procedure

Function	REBIND
ExternalSQL procedure name	MYSCHEMA.SQLPROC
Bind options	VALIDATE(RUN), ISOLATION(RR)
Return value	String returned in varying-length host variable returnval

The CALL statement is:

```
EXEC SQL CALL SYSPROC.DSNTPSMP('REBIND','MYSCHEMA.SQLPROC','',
'VALIDATE(RUN),ISOLATION(RR)', '', '', '', '',
'', '', '', '',
:returnval);
```

Result set that the SQL procedure processor (DSNTPSMP) returns:

DSNTPSMP returns one result set that contains messages and listings. You can write your client program to retrieve information from this result set. Because DSNTPSMP is a stored procedure, use the same technique that you would use to write a program to receive result sets from any stored procedure.

Each row of the result set contains the following information:

Processing step

The step in the DSNTPSMP *function* process to which the message applies.

DD name

The DD statement that identifies the data set that contains the message.

Sequence number

The sequence number of a line of message text within a message.

Message

A line of message text.

Rows in the message result set are ordered by processing step, DD name, and sequence number.

For an example of how to process a result set from DSNTPSMP, see the DB2 sample program DSNTJE65.

Related concepts:

“DB2 for z/OS SQL procedure processor (DSNTPSMP)” on page 602

 Job DSNTJE65 (DB2 Installation and Migration)

Related tasks:

“Writing a program to receive the result sets from a stored procedure” on page 807

Creating an external SQL procedure by using JCL

Using JCL is one of several ways that you can create and prepare an external SQL procedure.

Restriction: You cannot use JCL to prepare an external SQL procedure for debugging with the DB2 stored procedure debugger or the Unified Debugger. If you plan to use either of these debugging tools, use either DSNTPSMP or IBM Data Studio to create the external SQL procedure.

To create an external SQL procedure by using JCL, include the following job steps in your JCL job:

1. Issue a CREATE PROCEDURE statement that includes either the FENCED keyword or the EXTERNAL keyword and the procedure body, which is written in SQL.

Alternatively, you can issue the CREATE PROCEDURE statement dynamically by using an application such as SPUFI, DSNTEP2, DSNTIAD, or the command line processor.

Tip: If the routine body of the CREATE PROCEDURE statement contains embedded semicolons, change the default SQL terminator character from a semicolon to some other special character, such as the percent sign (%).

This statement defines the stored procedure to DB2. DB2 stores the definition in the DB2 catalog.

2. Run program DSNHPC with the HOST(SQL) option.

This program converts the external SQL procedure source statements into a C language program. DSNHPC also writes a new CREATE PROCEDURE statement in the data set that is specified in the SYSUT1 DD statement.

3. Precompile, compile, and link-edit the generated C program by using one of the following techniques:

- The DB2 precompiler and JCL instructions to compile and link-edit the program
- The SQL statement coprocessor

When you perform this step, specify the following settings:

- Give the DBRM the same name as the name of the load module for the external SQL procedure.
- Specify MARGINS(1,80) for the MARGINS SQL processing option.
- Specify the NOSEQ compiler option.

This process produces an executable C language program.

4. Bind the resulting DBRM into a package.

Suppose that you define an external SQL procedure by issuing the following CREATE PROCEDURE statement dynamically:

```
CREATE PROCEDURE DEVL7083.EMPDTLSS
(
  IN PEMPNO          CHAR(6)
  ,OUT PFIRSTNAME   VARCHAR(12)
  ,OUT PMIDINIT      CHAR(1)
  ,OUT PLASTNAME     VARCHAR(15)
  ,OUT PWORKDEPT    CHAR(3)
  ,OUT PHIREDATE     DATE
  ,OUT PSALARY       DEC(9,2)
```

```

,OUT PSQLCODE      INTEGER
)
RESULT SETS 0
MODIFIES SQL DATA
FENCED
NO DBINFO
WLM ENVIRONMENT DB9AWLMR
STAY RESIDENT NO
COLLID DEVL7083
PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF) '
COMMIT ON RETURN NO
LANGUAGE SQL
BEGIN
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
SELECT
    FIRSTNME
    , MIDINIT
    , LASTNAME
    , WORKDEPT
    , HIREDATE
    , SALARY
INTO PFIRSTNME
    , PMIDINIT
    , PLASTNAME
    , PWORKDEPT
    , PHIREDATE
    , PSALARY
FROM EMP
WHERE EMPNO = PEMPNO
;
DECLARE EXIT HANDLER FOR SQLEXCEPTION SET PSQLCODE = SQLCODE;
END

```

You can use JCL that is similar to the following JCL to prepare the procedure:

```

//ADMF001S JOB (999,POK),'SQL C/L/B/E',CLASS=A,MSGCLASS=T,          00000001
// NOTIFY=ADMF001,TIME=1440,REGION=0M                            00000002
/*JOBPARM SYSAFF=SC63,L=9999                                     00000003
// JCLLIB ORDER=(DB9AU.PROCLIB)                                  00000004
/*                                                                00250000
//JOBLIB DD DSN=DB9A9.SDSNEXIT,DISP=SHR                          00260000
//          DD DSN=DB9A9.SDSNLOAD,DISP=SHR                       00270000
//          DD DSN=CEE.SCEERUN,DISP=SHR                          00270001
/*-----                                                        00900000
/*          STEP 01: PRECOMP, COMP, LKED AN SQL PROCEDURE        01080000
/*-----                                                        00900000
//SQL01 EXEC DSNHSQL,MEM=EMPTLSS,
//          PARM.PC='HOST(SQL),SOURCE,XREF,MAR(1,80),STDSQL(NO)',
//          PARM.PCC='HOST(C),SOURCE,XREF,MAR(1,80),STDSQL(NO)',
//          PARM.C='SOURCE LIST MAR(1,80) NOSEQ LO RENT',
//          PARM.LKED='AMODE=31,RMODE=ANY,MAP,RENT'
//PC.SYSLIB DD DUMMY
//PC.SYSUT2 DD DSN=&&SPDML,DISP=(,PASS), &1t;=MAKE IT PERMANENT, IF YOU
//          UNIT=SYSDA,SPACE=(TRK,1),          WANT TO USE IT LATER
//          DCB=(RECFM=FB,LRECL=80)
//PC.SYSIN DD DISP=SHR,DSN=SG247083.PROD.DDL(&MEM.)
//PC.SYSCIN DD DISP=SHR,DSN=SG247083.TEST.C.SOURCE(&MEM.)
//PCC.SYSIN DD DISP=SHR,DSN=SG247083.TEST.C.SOURCE(&MEM.)
//PCC.SYSLIB DD DUMMY
//PCC.DBRMLIB DD DISP=SHR,DSN=SG247083.DEVL.DBRM(&MEM.)
//LKED.SYSLMOD DD DISP=SHR,DSN=SG247083.DEVL.LOAD(&MEM.)
//LKED.SYSIN DD * INCLUDE SYSLIB(DSNRLI)          NAME EMPTLSS(R)
/*
/*-----                                                        00900000
/*          STEP 02: BIND THE PROGRAM                             01290000

```

```

/*----- 01280000
//SQL02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT) 01300000
//DBRMLIB DD DSN=SG247083.DEVL.DBRM,DISP=SHR 01310000
//SYSTSPRT DD SYSOUT=* 01320000
//SYSPRINT DD SYSOUT=* 01330000
//SYSUDUMP DD SYSOUT=* 01340000
//SYSOUT DD SYSOUT=* 01350000
//REPORT DD SYSOUT=* 01360000
//SYSIN DD * 01370000
//SYSTSIN DD * 01390000
DSN SYSTEM(DB9A) 01400000
BIND PACKAGE (DEVL7083) MEMBER(EMPDTLSS) VALIDATE(BIND) - 01410000
OWNER(DEVL7083)
END 01460000
/* 01550000

```

Related concepts:

“SQL procedure body” on page 566

 Command line processor (DB2 Commands)

Related tasks:

“Changing SPUIFI defaults” on page 1041

“Creating an external SQL procedure by using DSNTPSMP” on page 601

 Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio)

Related reference:

“Descriptions of SQL processing options” on page 932

“DSNTEP2 and DSNTEP4” on page 1108

“DSNTIAD” on page 1106

 BIND PACKAGE (DSN) (DB2 Commands)

 CREATE PROCEDURE (SQL - external) (DB2 SQL)

Sample programs to help you prepare and run external SQL procedures

DB2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data set DSN10.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

See the prolog of each sample for specific instructions.

The following table lists the sample jobs that DB2 provides for external SQL procedures.

Table 99. External SQL procedure samples shipped with DB2

Member that contains source code	Contents	Purpose
DSNHSQL	JCL procedure	Precompiles, compiles, prelink-edits, and link-edits an external SQL procedure
DSNTEJ63	JCL job	Invokes JCL procedure DSNHSQL to prepare external SQL procedure DSN8ES1 for execution
DSN8ES1	External SQL procedure	A stored procedure that accepts a department number as input and returns a result set that contains salary information for each employee in that department
DSNTEJ64	JCL job	Prepares client program DSN8ED3 for execution

Table 99. External SQL procedure samples shipped with DB2 (continued)

Member that contains source code	Contents	Purpose
DSN8ED3	C program	Calls SQL procedure DSN8ES1
DSN8ES2	External SQL procedure	A stored procedure that accepts one input parameter and returns two output parameters. The input parameter specifies a bonus to be awarded to managers. The external SQL procedure updates the BONUS column of DSNA10.SDSNSAMP. If no SQL error occurs when the external SQL procedure runs, the first output parameter contains the total of all bonuses awarded to managers and the second output parameter contains a null value. If an SQL error occurs, the second output parameter contains an SQLCODE.
DSN8ED4	C program	Calls the SQL procedure processor, DSNTPSMP, to prepare DSN8ES2 for execution
DSN8WLMP	JCL procedure	A sample startup procedure for the WLM-established stored procedures address space in which DSNTPSMP runs
DSN8ED5	C program	Calls external SQL procedure DSN8ES2
DSNTEJ65	JCL job	Prepares and executes programs DSN8ED4 and DSN8ED5. DSNTEJ65 uses DSNTPSMP, the SQL procedure processor, which requires that the default EBCDIC CCSID that is used by DB2 also be compatible with the C compiler. Do not run DSNTEJ65 if the default EBCDIC CCSID for DB2 is not compatible with the C compiler. Examples of incompatible CCSIDs include 290, 930, 1026, and 1155.
DSNTEJ67	JCL job	Prepares an existing external SQL procedure (sample DSN8.DSN8ES2) for conversion to a native SQL procedure. DSNTEJ67 obtains the source of external SQL procedure DSN8.DSN8ES2 from the catalog and formats it into a data set. DSNTEJ67 executes DSNHPSM with HOST(SQLPL), obtains a listing for the source, and replaces the offending procedure options in the source data set.
DSNTIJRT	JCL job	Prepares a DB2 for z/OS server for operation with the SQL Debugger and the Unified Debugger

Creating an external stored procedure

An *external stored procedure* is a procedure that is written in a host language and can contain SQL statements. The source code for external procedures is separate from the definition.

Before you create an external procedure, Configure DB2 for running stored procedures and user-defined functions during installation or Configure DB2 for running stored procedures and user-defined functions during migration (DB2 Installation Guide).

Restriction: These instructions do not apply to Java stored procedures. The process for creating a Java stored procedure is different. The preparation process varies depending on what the procedure contains.

To create an external stored procedure:

1. Write the external stored procedure body in assembler, C, C++, COBOL, REXX, or PL/I.

Ensure that the procedure body that you write follows the guidelines for external stored procedures that are described in the following information:

- “Accessing other sites in an external procedure” on page 639
- “Accessing non-DB2 resources in your stored procedure” on page 639
- “Writing an external procedure to access IMS databases” on page 640
- “Writing an external procedure to return result sets to a distributed client” on page 641
- “Restrictions when calling other programs from an external stored procedure” on page 642
- “External stored procedures as main programs and subprograms” on page 644
- “Data types in stored procedures” on page 646
- “COMMIT and ROLLBACK statements in a stored procedure” on page 571

Restrictions:

- Do not include explicit attachment facility calls. External stored procedures that run in a WLM-established address space use Resource Recovery Services attachment facility (RRSAF) calls implicitly. If an external stored procedure makes an explicit attachment facility call, DB2 rejects the call.
- Do not include SRRCMIT or SRRBACK service calls. If an external stored procedure invokes either SRRCMIT or SRRBACK, DB2 puts the transaction in a state where a rollback operation is required and the CALL statement fails.

For REXX procedures, continue with step 3 on page 617.

2. For assembler, C, C++, COBOL, or PL/I stored procedures, prepare the external procedure by completing the following tasks:
 - a. Precompile, compile, and link-edit the application by using one of the following techniques:
 - The DB2 precompiler and JCL instructions to compile and link-edit the program
 - The SQL statement coprocessor

Recommendation: Compile and link-edit code as reentrant.

Link-edit the application by using DSNRLI, the language interface module for the Resource Recovery Services attachment facility, or DSNULI, the Universal language interface module. You must specify the parameter AMODE(31) when you link-edit the application with either of these modules. (24-bit applications are not supported.)

If you want to make the stored procedure reentrant, see “Creating an external stored procedure as reentrant” on page 643

If you want to run your procedure as a z/OS-authorized program, you must also perform the following tasks when you link-edit the application:

- Indicate that the load module can use restricted system services by specifying the parameter value AC=1.

- Put the load module for the stored procedure in an APF-authorized library.

You can compile COBOL stored procedures with either the DYNAM or NODYNAM COBOL compiler options. If you use DYNAM, ensure that the correct DB2 language interface module is loaded dynamically by performing one of the following actions:

- Specify the ATTACH(RRSAF) SQL processing option.
- Copy the DSNRLI module into a load library that is concatenated in front of the DB2 libraries. Use the member name DSNHLI.

- b. Bind the DBRM into a DB2 package by issuing the BIND PACKAGE command.

If you want to control access to a stored procedure package, specify the ENABLE bind option with the system connection type of the calling application.

Stored procedures require only a package. You do not need to bind a plan for the stored procedure or bind the stored procedure package to the plan for the calling application. For remote access scenarios, you need a package at both the requester and server sites.

For more information about stored procedure packages, see “Packages and plans for external stored procedures” on page 637. The following example BIND PACKAGE command binds the DBRM EMPDTL1P to the collection DEVL7083.

```

BIND PACKAGE(DEVL7083) -
MEMBER(EMPDTL1P) ACT(REP) ISO(UR) ENCODING(EBCDIC) -
OWNER(DEVL7083) LIBRARY('SG247083.DEVL.DBRM')

```

3. Define the stored procedure to DB2 by issuing the CREATE PROCEDURE statement with the EXTERNAL option. Use the EXTERNAL NAME clause to specify the name of the load module for the program that runs when this procedure is called.

If you want to run your procedure as a z/OS-authorized program, specify an appropriate environment with the WLM ENVIRONMENT option. The stored procedure must run in an address space with a startup procedure in which all libraries in the STEPLIB concatenation are APF-authorized.

If you want environment information to be passed to the stored procedure when it is invoked, specify the DBINFO and PARAMETER STYLE SQL options in the CREATE PROCEDURE statement. When the procedure is invoked, DB2 passes the DBINFO structure, which contains environment information, to the stored procedure. For more information about PARAMETER STYLE, see “Defining the linkage convention for an external stored procedure” on page 619.

If you compiled the stored procedure as reentrant, specify the STAY RESIDENT YES option in the CREATE PROCEDURE statement. This option makes the procedure remain resident in storage.

4. Authorize the appropriate users to use the stored procedure by issuing the GRANT EXECUTE statement.

Example: The following statement allows an application that runs under the authorization ID JONES to call stored procedure SPSHEMA.STORPRCA:

```

GRANT EXECUTE ON PROCEDURE SPSHEMA.STORPRCA TO JONES;

```

Example of defining a C stored procedure

Suppose that you have written and prepared a stored procedure that has the following characteristics:

- The name of the stored procedure is B.
- The stored procedure has the following two parameters:
 - An integer input parameter that is named V1
 - A character output parameter of length 9 that is named V2
- The stored procedure is written in the C language.
- The stored procedure contains no SQL statements.
- The same input always produces the same output.
- The load module name is SUMMOD.
- The package collection name is SUMCOLL.
- The stored procedure is to run for no more than 900 CPU service units.
- The parameters can have null values.
- The stored procedure is to be deleted from memory when it completes.
- The stored procedure needs the following Language Environment runtime options:
MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)
- The stored procedure is part of the WLM application environment that is named PAYROLL.
- The stored procedure runs as a main program.
- The stored procedure does not access non-DB2 resources, so it does not need a special RACF environment.
- The stored procedure can return at most 10 result sets.
- When control returns to the client program, DB2 does not commit updates automatically.

The following CREATE PROCEDURE statement defines the stored procedure to DB2:

```
CREATE PROCEDURE B(IN V1 INTEGER, OUT V2 CHAR(9))
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME SUMMOD
COLLID SUMCOLL
ASUTIME LIMIT 900
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON) '
WLM ENVIRONMENT PAYROLL
PROGRAM TYPE MAIN
SECURITY DB2
DYNAMIC RESULT SETS 10
COMMIT ON RETURN NO;
```

You can now invoke the stored procedure from an application program or command line processor.

Related concepts:

“Universal language interface” on page 153

 [Java stored procedures and user-defined functions \(DB2 Application Programming for Java\)](#)

Related tasks:

➤ [Implementing DB2 stored procedures \(DB2 Administration Guide\)](#)

Related reference:

➤ [BIND and REBIND options for packages and plans \(DB2 Commands\)](#)

➤ [CREATE PROCEDURE \(external\) \(DB2 SQL\)](#)

➤ [GRANT \(function or procedure privileges\) \(DB2 SQL\)](#)

➤ [C programming \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

➤ [COBOL programming \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

➤ [Four release levels: Sample scenario \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

➤ [REXX programming \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

Defining the linkage convention for an external stored procedure

A linkage convention specifies the rules for the parameter list that is passed by the program that calls the external stored procedure. For example, the convention can specify whether the calling program can pass null values for input parameters.

To define the linkage convention for a stored procedure:

When you define the stored procedure with the CREATE PROCEDURE statement, specify one of the following values for the PARAMETER STYLE option:

- GENERAL
- GENERAL WITH NULLS
- SQL

SQL is the default.

Linkage conventions for external stored procedures:

The linkage convention for a stored procedure can be either GENERAL, GENERAL WITH NULLS, or SQL. These linkage conventions apply to only external stored procedures.

GENERAL

Specify the GENERAL linkage convention when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure. If you specify GENERAL, ensure that the stored procedure contains a variable declaration for each parameter that is passed in the CALL statement.

The following figure shows the structure of the parameter list for PARAMETER STYLE GENERAL.

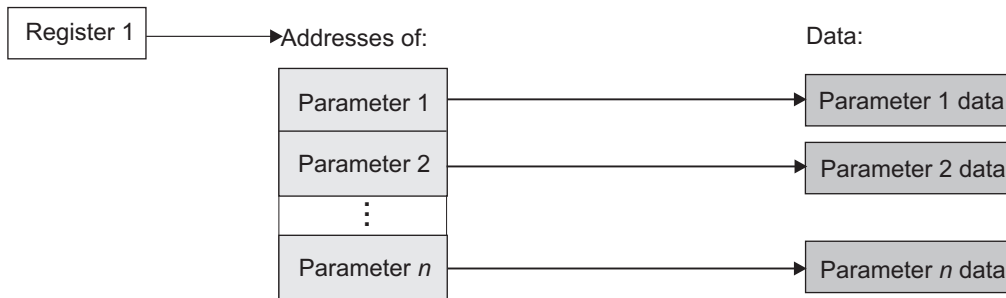


Figure 33. Parameter convention *GENERAL* for a stored procedure

GENERAL WITH NULLS

Specify the *GENERAL WITH NULLS* linkage convention when you want to allow the calling program to supply a null value for any parameter that is passed to the stored procedure. If you specify *GENERAL WITH NULLS*, ensure that the stored procedure performs the following tasks:

- Declares a variable for each parameter that is passed in the *CALL* statement.
- Declares a null indicator structure that contains an indicator variable for each parameter.
- On entry, examines all indicator variables that are associated with input parameters to determine which parameters contain null values.
- On exit, assigns values to all indicator variables that are associated with output variables. If the output variable returns a null value to the caller, assign the associated indicator variable a negative number. Otherwise, assign a value of 0 to the indicator variable.

In the *CALL* statement in the calling application, follow each parameter with its indicator variable. Use one of the following forms:

- *host-variable :indicator-variable*
- *host-variable INDICATOR :indicator-variable*

The following figure shows the structure of the parameter list for *PARAMETER STYLE GENERAL WITH NULLS*.

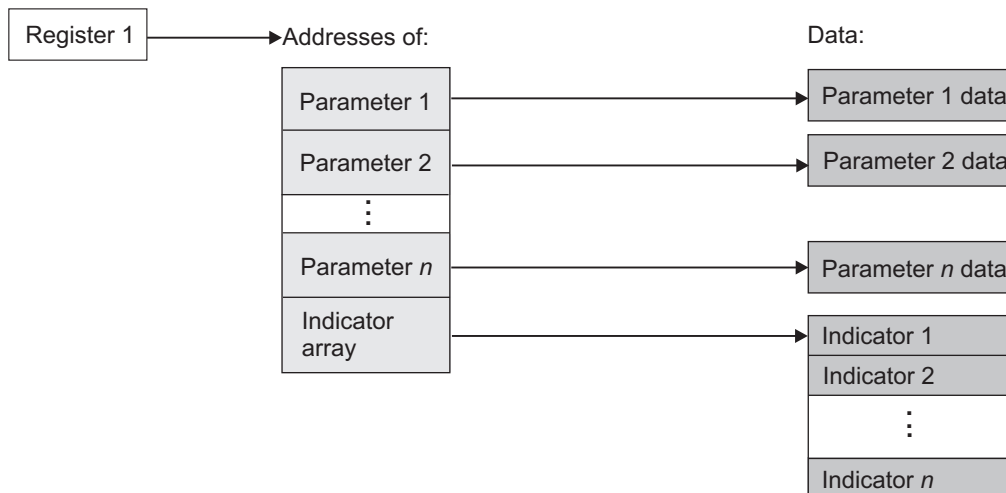


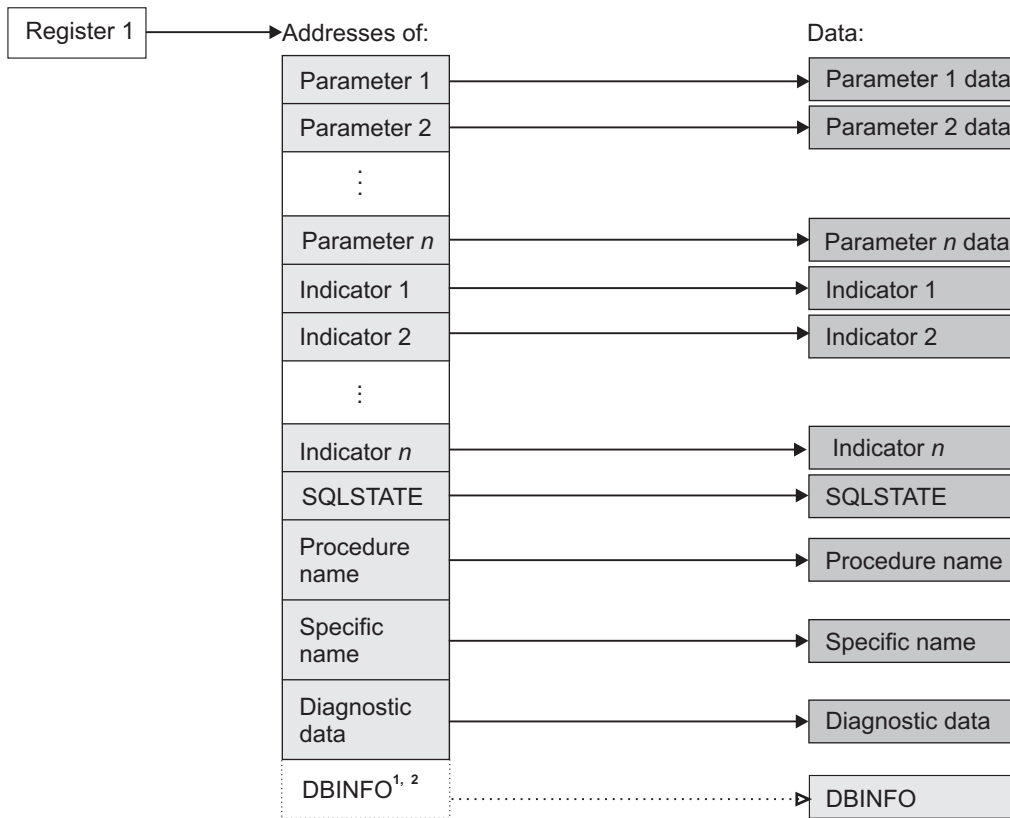
Figure 34. Parameter convention *GENERAL WITH NULLS* for a stored procedure

SQL Specify the SQL linkage convention when you want both of the following conditions:

- The calling program to be able to supply a null value for any parameter that is passed to the stored procedure.
- DB2 to pass input and output parameters to the stored procedure that contain the following information:
 - The SQLSTATE that is to be returned to DB2. This value is a CHAR(5) parameter that represents the SQLSTATE that is passed into the program from the database manager. The initial value is set to '00000'. Although the SQLSTATE is usually not set by the program, it can be set as the result SQLSTATE that is used to return an error or a warning. Returned values that start with anything other than '00', '01', or '02' are error conditions.
 - The qualified name of the stored procedure. This is a VARCHAR(128) value.
 - The specific name of the stored procedure. The specific name is a VARCHAR(128) value that is the same as the unqualified name.
 - The SQL diagnostic string that is to be returned to DB2. This is a VARCHAR(1000) value. Use this area to pass descriptive information about an error or warning to the caller.

Restriction: You cannot use the SQL linkage convention for a REXX language stored procedure.

The following figure shows the structure of the parameter list for PARAMETER STYLE SQL.



¹ For PL/I, this value is the address of a pointer to the DBINFO data.
² Passed if the DBINFO option is specified in the user-defined function definition

Figure 35. Parameter convention SQL for a stored procedure

Related concepts:

“Examples of programs that call stored procedures” on page 252

Related reference:

- CREATE PROCEDURE (external) (DB2 SQL)
- SQLSTATE values and common error codes (DB2 Codes)

Example of GENERAL linkage convention:

Specify the GENERAL linkage convention when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure.

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL linkage convention to receive parameters.

For these examples, assume that a COBOL application has the following parameter declarations and CALL statement:

```
*****
* PARAMETERS FOR THE SQL STATEMENT CALL          *
*****
01 V1 PIC S9(9) USAGE COMP.
01 V2 PIC X(9).
```



```

:
EXEC SQL CALL A (:V1, :V2) END-EXEC.

```

In the CREATE PROCEDURE statement, the parameters are defined as follows:
IN V1 INT, OUT V2 CHAR(9)

Assembler example: The following example shows how a stored procedure that is written in assembler language receives these parameters.

```

*****
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES *
* THE GENERAL LINKAGE CONVENTION. *
*****
A CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
  USING PROGAREA,R13
*****
* BRING UP THE LANGUAGE ENVIRONMENT. *
*****

:
*****
* GET THE PASSED PARAMETER VALUES. THE GENERAL LINKAGE CONVENTION*
* FOLLOWS THE STANDARD ASSEMBLER LINKAGE CONVENTION: *
* ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS TO THE *
* PARAMETERS. *
*****
      L    R7,0(R1)          GET POINTER TO V1
      MVC  LOC1(4),0(R7)     MOVE VALUE INTO LOCAL COPY OF V1

:

      L    R7,4(R1)          GET POINTER TO V2
      MVC  0(9,R7),LOCV2    MOVE A VALUE INTO OUTPUT VAR V2

:

      CEETERM RC=0
*****
* VARIABLE DECLARATIONS AND EQUATES *
*****
R1      EQU 1                REGISTER 1
R7      EQU 7                REGISTER 7
PPA     CEEPPA ,            CONSTANTS DESCRIBING THE CODE BLOCK
        LTORG ,             PLACE LITERAL POOL HERE
PROGAREA DSECT
        ORG  **CEEDSASZ     LEAVE SPACE FOR DSA FIXED PART
LOCV1   DS  F                LOCAL COPY OF PARAMETER V1
LOCV2   DS  CL9             LOCAL COPY OF PARAMETER V2

:

PROGSIZE EQU  *-PROGAREA
        CEEDSA ,           MAPPING OF THE DYNAMIC SAVE AREA
        CEECAA ,           MAPPING OF THE COMMON ANCHOR AREA
        END  A

```

C example: The following figure shows how a stored procedure that is written in the C language receives these parameters.

```

#pragma runopts(PLIST(OS))
#pragma options(RENT)
#include <stdlib.h>
#include <stdio.h>
/*****
/* Code for a C language stored procedure that uses the */
/* GENERAL linkage convention. */
/*****
main(argc,argv)

```

```

int argc;                /* Number of parameters passed */
char *argv[];           /* Array of strings containing */
                        /* the parameter values      */
{
  long int locv1;        /* Local copy of V1          */
  char locv2[10];       /* Local copy of V2          */
                        /* (null-terminated)        */
:
:
  /*****
  /* Get the passed parameters. The GENERAL linkage convention */
  /* follows the standard C language parameter passing         */
  /* conventions:                                              */
  /* - argc contains the number of parameters passed           */
  /* - argv[0] is a pointer to the stored procedure name       */
  /* - argv[1] to argv[n] are pointers to the n parameters    */
  /* in the SQL statement CALL.                                */
  /*****
  if(argc==3)           /* Should get 3 parameters: */
  {                     /* procname, V1, V2         */
    locv1 = *(int *) argv[1];
                        /* Get local copy of V1    */
:
:
    strcpy(argv[2],locv2);
                        /* Assign a value to V2    */
:
:
  }
}

```

COBOL example: The following figure shows how a stored procedure that is written in the COBOL language receives these parameters.

```

CBL RENT
IDENTIFICATION DIVISION.
*****
* CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
* GENERAL LINKAGE CONVENTION.                               *
*****
PROGRAM-ID.    A.

:
:
DATA DIVISION.

:
:
LINKAGE SECTION.
*****
* DECLARE THE PARAMETERS PASSED BY THE SQL STATEMENT      *
* CALL HERE.                                              *
*****
01 V1 PIC S9(9) USAGE COMP.
01 V2 PIC X(9).

:
:
PROCEDURE DIVISION USING V1, V2.
*****
* THE USING PHRASE INDICATES THAT VARIABLES V1 AND V2    *
* WERE PASSED BY THE CALLING PROGRAM.                    *
*****
:
:
*****
* ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
*****
MOVE '123456789' TO V2.

```

PL/I example: The following figure shows how a stored procedure that is written in the PL/I language receives these parameters.

```
*PROCESS SYSTEM(MVS);
A: PROC(V1, V2) OPTIONS(MAIN NOEXECOPS REENTRANT);
/*****
/* Code for a PL/I language stored procedure that uses the */
/* GENERAL linkage convention. */
/*****
/* Indicate on the PROCEDURE statement that two parameters */
/* were passed by the SQL statement CALL. Then declare the */
/* parameters in the following section. */
/*****
DCL V1 BIN FIXED(31),
     V2 CHAR(9);

:
V2 = '123456789'; /* Assign a value to output variable V2 */
```

Example of GENERAL WITH NULLS linkage convention:

Specify the GENERAL WITH NULLS linkage convention when you want to allow the calling program to supply a null value for any parameter that is passed to the stored procedure.

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL WITH NULLS linkage convention to receive parameters.

For these examples, assume that a C application has the following parameter declarations and CALL statement:

```
/*****
/* Parameters for the SQL statement CALL */
/*****
long int v1;
char v2[10]; /* Allow an extra byte for */
             /* the null terminator */
/*****
/* Indicator structure */
/*****
struct indicators {
    short int ind1;
    short int ind2;
} indstruc;

:
indstruc.ind1 = 0; /* Remember to initialize the */
                  /* input parameter's indicator*/
                  /* variable before executing */
                  /* the CALL statement */
EXEC SQL CALL B (:v1 :indstruc.ind1, :v2 :indstruc.ind2);

:
```

In the CREATE PROCEDURE statement, the parameters are defined as follows:
IN V1 INT, OUT V2 CHAR(9)

Assembler example: The following figure shows how a stored procedure that is written in assembler language receives these parameters.

```

*****
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES *
* THE GENERAL WITH NULLS LINKAGE CONVENTION. *
*****
B CEEENTRY AUTO=PROG SIZE,MAIN=YES,PLIST=OS
  USING PROGAREA,R13
*****
* BRING UP THE LANGUAGE ENVIRONMENT. *
*****

:
:
*****
* GET THE PASSED PARAMETER VALUES. THE GENERAL WITH NULLS LINKAGE*
* CONVENTION IS AS FOLLOWS: *
* ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N *
* PARAMETERS ARE PASSED, THERE ARE N+1 POINTERS. THE FIRST *
* N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS *
* WITH THE GENERAL LINKAGE CONVENTION. THE N+1ST POINTER IS *
* THE ADDRESS OF A LIST CONTAINING THE N INDICATOR VARIABLE *
* VALUES. *
*****
      L   R7,0(R1)           GET POINTER TO V1
      MVC LOCV1(4),0(R7)     MOVE VALUE INTO LOCAL COPY OF V1
      L   R7,8(R1)           GET POINTER TO INDICATOR ARRAY
      MVC LOCIND(2*2),0(R7)  MOVE VALUES INTO LOCAL STORAGE
      LH  R7,LOCIND          GET INDICATOR VARIABLE FOR V1
      LTR R7,R7              CHECK IF IT IS NEGATIVE
      BM  NULLIN             IF SO, V1 IS NULL

:
:
      L   R7,4(R1)           GET POINTER TO V2
      MVC 0(9,R7),LOCV2     MOVE A VALUE INTO OUTPUT VAR V2
      L   R7,8(R1)           GET POINTER TO INDICATOR ARRAY
      MVC 2(2,R7),=H(0)     MOVE ZERO TO V2'S INDICATOR VAR

:
:
      CEETERM RC=0
*****
* VARIABLE DECLARATIONS AND EQUATES *
*****
R1 EQU 1 REGISTER 1
R7 EQU 7 REGISTER 7
PPA CEEPPA , CONSTANTS DESCRIBING THE CODE BLOCK
LTORG , PLACE LITERAL POOL HERE
PROGAREA DSECT
ORG ++CEEDSASZ LEAVE SPACE FOR DSA FIXED PART
LOCV1 DS F LOCAL COPY OF PARAMETER V1
LOCV2 DS CL9 LOCAL COPY OF PARAMETER V2
LOCIND DS 2H LOCAL COPY OF INDICATOR ARRAY

:
:
PROG SIZE EQU *-PROGAREA
CEEDSA , MAPPING OF THE DYNAMIC SAVE AREA
CEECAA , MAPPING OF THE COMMON ANCHOR AREA
END B

```

C example: The following figure shows how a stored procedure that is written in the C language receives these parameters.

```

#pragma options(RENT)
#pragma runopts(PLIST(OS))
#include <stdlib.h>
#include <stdio.h>
/*****
/* Code for a C language stored procedure that uses the */
/* GENERAL WITH NULLS linkage convention. */

```

```

/*****
main(argc,argv)
    int argc;                /* Number of parameters passed */
    char *argv[];           /* Array of strings containing */
                            /* the parameter values      */
{
    long int locv1;         /* Local copy of V1          */
    char locv2[10];        /* Local copy of V2          */
                            /* (null-terminated)        */
    short int locind[2];   /* Local copy of indicator   */
                            /* variable array           */
    short int *tempint;    /* Used for receiving the   */
                            /* indicator variable array  */
:
/*****
/* Get the passed parameters. The GENERAL WITH NULLS linkage */
/* convention is as follows: */
/* - argc contains the number of parameters passed          */
/* - argv[0] is a pointer to the stored procedure name      */
/* - argv[1] to argv[n] are pointers to the n parameters   */
/*   in the SQL statement CALL.                             */
/* - argv[n+1] is a pointer to the indicator variable array */
/*****
if(argc==4)              /* Should get 4 parameters: */
{                          /* procname, V1, V2,        */
                            /* indicator variable array */

    locv1 = *(int *) argv[1]; /* Get local copy of V1     */
    tempint = argv[3];       /* Get pointer to indicator */
                            /* variable array           */
    locind[0] = *tempint;    /* Get 1st indicator variable */
    locind[1] = *(++tempint); /* Get 2nd indicator variable */
    if(locind[0]<0)          /* If 1st indicator variable */
    {                          /* is negative, V1 is null  */
:
    }
:
:
    strcpy(argv[2],locv2);  /* Assign a value to V2     */
    *(++tempint) = 0;      /* Assign 0 to V2's indicator */
                            /* variable                 */
}
}

```

COBOL example: The following figure shows how a stored procedure that is written in the COBOL language receives these parameters.

```

CBL RENT
IDENTIFICATION DIVISION.
*****
* CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
* GENERAL WITH NULLS LINKAGE CONVENTION.                  *
*****
PROGRAM-ID.    B.
:
:
DATA DIVISION.
:
:
LINKAGE SECTION.
*****

```

```

* DECLARE THE PARAMETERS AND THE INDICATOR ARRAY THAT      *
* WERE PASSED BY THE SQL STATEMENT CALL HERE.             *
*****
01 V1 PIC S9(9) USAGE COMP.
01 V2 PIC X(9).
*
01 INDARRAY.
   10 INDVAR PIC S9(4) USAGE COMP OCCURS 2 TIMES.

:
PROCEDURE DIVISION USING V1, V2, INDARRAY.
*****
* THE USING PHRASE INDICATES THAT VARIABLES V1, V2, AND *
* INDARRAY WERE PASSED BY THE CALLING PROGRAM.          *
*****

:
*****
* TEST WHETHER V1 IS NULL *
*****
IF INDARRAY(1) < 0
   PERFORM NULL-PROCESSING.

:
*****
* ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
* AND ITS INDICATOR VARIABLE          *
*****
MOVE '123456789' TO V2.
MOVE ZERO TO INDARRAY(2).

```

PL/I example: The following figure shows how a stored procedure that is written in the PL/I language receives these parameters.

```

*PROCESS SYSTEM(MVS);
A: PROC(V1, V2, INDSTRUC) OPTIONS(MAIN NOEXECOPS REENTRANT);
/*****
/* Code for a PL/I language stored procedure that uses the */
/* GENERAL WITH NULLS linkage convention.                  */
/*****
/*****
/* Indicate on the PROCEDURE statement that two parameters */
/* and an indicator variable structure were passed by the SQL */
/* statement CALL. Then declare them in the following section.*/
/* For PL/I, you must declare an indicator variable structure, */
/* not an array.                                           */
/*****
DCL V1 BIN FIXED(31),
     V2 CHAR(9);
DCL
   01 INDSTRUC,
     02 IND1 BIN FIXED(15),
     02 IND2 BIN FIXED(15);

:
IF IND1 < 0 THEN
   CALL NULLVAL; /* If indicator variable is negative */
                /* then V1 is null                    */

:
V2 = '123456789'; /* Assign a value to output variable V2 */
IND2 = 0;         /* Assign 0 to V2's indicator variable */

```



```

* THE ADDRESSES OF THE INDICATOR VARIABLE VALUES. THE LAST
*
* 4 POINTERS (5, IF DBINFO IS PASSED) ARE THE ADDRESSES OF
*
* INFORMATION ABOUT THE STORED PROCEDURE ENVIRONMENT AND
*
* EXECUTION RESULTS.
*
*****
      L   R7,0(R1)           GET POINTER TO V1
      MVC LOCV1(4),0(R7)     MOVE VALUE INTO LOCAL COPY OF V1
      L   R7,8(R1)           GET POINTER TO 1ST INDICATOR VARIABLE
      MVC LOCI1(2),0(R7)     MOVE VALUE INTO LOCAL STORAGE
      L   R7,20(R1)          GET POINTER TO STORED PROCEDURE
NAME
      MVC LOCSPNM(20),0(R7)  MOVE VALUE INTO LOCAL STORAGE
      L   R7,24(R1)          GET POINTER TO DBINFO
      MVC LOCDBINF(DBINFLN),0(R7)
*
      LH  R7,LOCI1           GET INDICATOR VARIABLE FOR V1
      LTR R7,R7              CHECK IF IT IS NEGATIVE
      BM  NULLIN             IF SO, V1 IS NULL

:
      L   R7,4(R1)           GET POINTER TO V2
      MVC 0(9,R7),LOCV2      MOVE A VALUE INTO OUTPUT VAR V2
      L   R7,12(R1)          GET POINTER TO INDICATOR VAR 2
      MVC 0(2,R7),=H'0'      MOVE ZERO TO V2'S INDICATOR VAR
      L   R7,16(R1)          GET POINTER TO SQLSTATE
      MVC 0(5,R7),=CL5'xxxxx' MOVE xxxxx TO SQLSTATE

:
      CEETERM RC=0
*****
* VARIABLE DECLARATIONS AND EQUATES
*
*****
R1      EQU 1                REGISTER 1
R7      EQU 7                REGISTER 7
PPA     CEEPPA ,             CONSTANTS DESCRIBING THE CODE BLOCK
        LTORG ,              PLACE LITERAL POOL HERE
PROGAREA DSECT
        ORG  **CEEDSASZ      LEAVE SPACE FOR DSA FIXED PART
LOCV1   DS  F                LOCAL COPY OF PARAMETER V1
LOCV2   DS  CL9              LOCAL COPY OF PARAMETER V2
LOCI1   DS  H                LOCAL COPY OF INDICATOR 1
LOCI2   DS  H                LOCAL COPY OF INDICATOR 2
LOCSQST DS  CL5              LOCAL COPY OF SQLSTATE
LOCSPNM DS  H,CL27           LOCAL COPY OF STORED PROC NAME
LOCSPSNM DS H,CL18           LOCAL COPY OF SPECIFIC NAME
LOCDIAG DS  H,CL1000         LOCAL COPY OF DIAGNOSTIC DATA
LOCDBINF DS  0H              LOCAL COPY OF DBINFO DATA
DBNAMELN DS  H                DATABASE NAME LENGTH
DBNAME  DS  CL128            DATABASE NAME
AUTHIDLN DS  H                APPL AUTH ID LENGTH
AUTHID  DS  CL128            APPL AUTH ID
ASC_SBCS DS  F                ASCII SBCS CCSID
ASC_DBCS DS  F                ASCII DBCS CCSID
ASC_MIXD DS  F                ASCII MIXED CCSID
EBC_SBCS DS  F                EBCDIC SBCS CCSID
EBC_DBCS DS  F                EBCDIC DBCS CCSID
EBC_MIXD DS  F                EBCDIC MIXED CCSID
UNI_SBCS DS  F                UNICODE SBCS CCSID
UNI_DBCS DS  F                UNICODE DBCS CCSID
UNI_MIXD DS  F                UNICODE MIXED CCSID
ENCOD  DS  F                PROCEDURE ENCODING SCHEME

```



```

RESERV0 DS    CL20                RESERVED
TBQUALLN DS   H                   TABLE QUALIFIER LENGTH
TBQUAL    DS   CL128              TABLE QUALIFIER
TBNAMELN  DS   H                   TABLE NAME LENGTH
TBNAME    DS   CL128              TABLE NAME
CLNAMELN  DS   H                   COLUMN NAME LENGTH
COLNAME   DS   CL128              COLUMN NAME
RELVER    DS   CL8                 DBMS RELEASE AND VERSION
RESERV1   DS   CL2                 RESERVED
PLATFORM  DS   F                   DBMS OPERATING SYSTEM
NUMTFCOL  DS   H                   NUMBER OF TABLE FUNCTION COLS USED
RESERV2   DS   CL26               RESERVED
TFCOLNUM  DS   A                   POINTER TO TABLE FUNCTION COL LIST
APPLID    DS   A                   POINTER TO APPLICATION ID
RESERV3   DS   CL20               RESERVED
DBINFLN   EQU  *-LOCDBINF          LENGTH OF DBINFO

:
PROGSIZE  EQU  *-PROGAREA
          CEEDSA ,                 MAPPING OF THE DYNAMIC SAVE AREA
          CEECAA ,                 MAPPING OF THE COMMON ANCHOR AREA
          END    B

```

C example: The following figure shows how a stored procedure that is written as a main program in the C language receives these parameters.

```

#pragma runopts(plist(os))
#include <stdlib.h>
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
    int parm1;
    short int ind1;
    char p_proc[28];
    char p_spec[19];
    /*****
    /* Assume that the SQL CALL statement included
    /* 3 input/output parameters in the parameter list.*/
    /* The argv vector will contain these entries:
    /*   argv[0]          1  contains load module
    /*   argv[1-3]       3  input/output parms
    /*   argv[4-6]       3  null indicators
    /*   argv[7]         1  SQLSTATE variable
    /*   argv[8]         1  qualified proc name
    /*   argv[9]         1  specific proc name
    /*   argv[10]        1  diagnostic string
    /*   argv[11]        + 1  dbinfo
    /*                   -----
    /*                   12  for the argc variable
    *****/
    if argc<>12 {

:
    /* We end up here when invoked with wrong number of parms */
    }

    /*****
    /* Assume the first parameter is an integer.
    /* The following code shows how to copy the integer*/
    /* parameter into the application storage.
    *****/
    parm1 = *(int *) argv[1];
    /*****
    /* We can access the null indicator for the first

```

```

/* parameter on the SQL CALL as follows:          */
/*****/
ind1 = *(short int *) argv[4];
/*****/
/* We can use the following expression           */
/* to assign 'xxxxx' to the SQLSTATE returned to */
/* caller on the SQL CALL statement.            */
/*****/
strcpy(argv[7],"xxxxx/0");
/*****/
/* We obtain the value of the qualified procedure */
/* name with this expression.                    */
/*****/
strcpy(p_proc,argv[8]);
/*****/
/* We obtain the value of the specific procedure */
/* name with this expression.                    */
/*****/
strcpy(p_spec,argv[9]);
/*****/
/* We can use the following expression to assign */
/* 'yyyyyyy' to the diagnostic string returned   */
/* in the SQLDA associated with the CALL statement.*/
/*****/
strcpy(argv[10],"yyyyyyy/0");
:
}

```

The following figure shows how a stored procedure that is written as a subprogram in the C language receives these parameters.

```

#pragma linkage(myproc,fetchable)
#include <stdlib.h>
#include <stdio.h>
#include <sqludf.h>

void myproc(*parm1 int,          /* assume INT for PARM1
*/
           parm2 char[11],      /* assume CHAR(10) parm2
*/
           :
           *p_ind1 short int,   /* null indicator for parm1
*/
           *p_ind2 short int,   /* null indicator for parm2
*/
           :
           p_sqlstate char[6],  /* SQLSTATE returned to DB2
*/
           p_proc char[28],     /* Qualified stored proc name
*/
           p_spec char[19],     /* Specific stored proc name
*/
           p_diag char[1001],   /* Diagnostic string
*/
           struct sqludf_dbinfo *udf_dbinfo); /* DBINFO
*/
{
  int l_p1;
  char[l_p1] l_p2;
  short int l_ind1;
  short int l_ind2;
  char[6] l_sqlstate;
  char[28] l_proc;
  char[19] l_spec;
  char[71] l_diag;
  sqludf_dbinfo *ludf_dbinfo;
}

```

```

:
:
/*****
/* Copy each of the parameters in the parameter */
/* list into a local variable, just to demonstrate */
/* how the parameters can be referenced. */
/*****
l_p1 = *parm1;

strcpy(l_p2,parm2);

l_ind1 = *p_ind1;

l_ind1 = *p_ind2;

strcpy(l_sqlstate,p_sqlstate);

strcpy(l_proc,p_proc);

strcpy(l_spec,p_spec);

strcpy(l_diag,p_diag);
memcpy(&ludf_dbinfo,udf_dbinfo,sizeof(ludf_dbinfo));
:
:
}

```

COBOL example: The following figure shows how a stored procedure that is written in the COBOL language receives these parameters.

```

CBL RENT
IDENTIFICATION DIVISION.
:
:
DATA DIVISION.
:
:
LINKAGE SECTION.
* Declare each of the parameters
01 PARM1 ...
01 PARM2 ...
:
:
* Declare a null indicator for each parameter
01 P-IND1 PIC S9(4) USAGE COMP.
01 P-IND2 PIC S9(4) USAGE COMP.
:
:
* Declare the SQLSTATE that can be set by stored proc
01 P-SQLSTATE PIC X(5).
* Declare the qualified procedure name
01 P-PROC.
    49 P-PROC-LEN PIC 9(4) USAGE BINARY.
    49 P-PROC-TEXT PIC X(27).
* Declare the specific procedure name
01 P-SPEC.
    49 P-SPEC-LEN PIC 9(4) USAGE BINARY.
    49 P-SPEC-TEXT PIC X(18).
* Declare SQL diagnostic message token
01 P-DIAG.
    49 P-DIAG-LEN PIC 9(4) USAGE BINARY.
    49 P-DIAG-TEXT PIC X(1000).
*****
* Structure used for DBINFO *
*****
01 SQLUDF-DBINFO.
*   Location name length
    05 DBNAMELEN PIC 9(4) USAGE BINARY.
*   Location name
    05 DBNAME PIC X(128).
*   authorization ID length
    05 AUTHIDLEN PIC 9(4) USAGE BINARY.

```



```

DCL 01 P_PROC CHAR(27) /* Qualified procedure name */
VARYING;
DCL 01 P_SPEC CHAR(18) /* Specific stored proc */
VARYING;
DCL 01 P_DIAG CHAR(1000) /* Diagnostic string */
VARYING;
DCL DBINFO PTR;
DCL 01 SP_DBINFO BASED(DBINFO),
/* Dbinfo */
03 UDF_DBINFO_LLEN BIN FIXED(15), /* location length */
03 UDF_DBINFO_LOC CHAR(128), /* location name */
03 UDF_DBINFO_ALEN BIN FIXED(15), /* auth ID length */
03 UDF_DBINFO_AUTH CHAR(128), /* authorization ID */
03 UDF_DBINFO_CCSID, /* CCSIDs for DB2 for z/OS */
05 R1 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_ASBCS BIN FIXED(15), /* ASCII SBCS CCSID */
05 R2 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_ADBCS BIN FIXED(15), /* ASCII DBCS CCSID */
05 R3 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_AMIXED BIN FIXED(15), /* ASCII MIXED CCSID */
05 R4 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_ESBCS BIN FIXED(15), /* EBCDIC SBCS CCSID */
05 R5 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_EDBCS BIN FIXED(15), /* EBCDIC DBCS CCSID */
05 R6 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_EMIXED BIN FIXED(15), /* EBCDIC MIXED CCSID*/
05 R7 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_USBCS BIN FIXED(15), /* Unicode SBCS CCSID
*/
05 R8 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_UDBCS BIN FIXED(15), /* Unicode DBCS CCSID
*/
05 R9 BIN FIXED(15), /* Reserved */
05 UDF_DBINFO_UMIXED BIN FIXED(15), /* Unicode MIXED CCSID*/
05 UDF_DBINFO_ENCODE BIN FIXED(31), /* SP encode scheme */
05 UDF_DBINFO_RESERV0 CHAR(20), /* reserved
*/
03 UDF_DBINFO_SLEN BIN FIXED(15), /* schema length */
03 UDF_DBINFO_SCHEMA CHAR(128), /* schema name */
03 UDF_DBINFO_TLEN BIN FIXED(15), /* table length */
03 UDF_DBINFO_TABLE CHAR(128), /* table name */
03 UDF_DBINFO_CLEN BIN FIXED(15), /* column length */
03 UDF_DBINFO_COLUMN CHAR(128), /* column name */
03 UDF_DBINFO_RELVER CHAR(8), /* DB2 release level */
03 UDF_DBINFO_RESERV0 CHAR(2), /* reserved */
03 UDF_DBINFO_PLATFORM BIN FIXED(31), /* database platform*/
03 UDF_DBINFO_NUMTFCOL BIN FIXED(15), /* # of TF cols used*/
03 UDF_DBINFO_RESERV1 CHAR(26), /* reserved */
03 UDF_DBINFO_TFCOLUMN PTR, /* -> table fun col list */
03 UDF_DBINFO_APPLID PTR, /* -> application id */
03 UDF_DBINFO_RESERV2 CHAR(20); /* reserved */
:

```

DBINFO structure

Use the DBINFO structure to pass environment information to user-defined functions and stored procedures. Some fields in the structure are not used for stored procedures.

DBINFO 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.

The DBINFO structure includes the following information:

Location name length

An unsigned 2-byte integer field. It contains the length of the location name in the next field.

Location name

A 128-byte character field. It contains the name of the location to which the invoker is currently connected.

Authorization ID length

An unsigned 2-byte integer field. It contains the length of the authorization ID in the next field.

Authorization ID

A 128-byte character field. It contains the authorization ID of the application from which the stored procedure is invoked, padded on the right with blanks. If this stored procedure is nested within other routines (user-defined functions or stored procedures), this value is the authorization ID of the application that invoked the highest-level routine.

Subsystem code page

A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs of the subsystem from which the stored procedure is invoked.

Table qualifier length

An unsigned 2-byte integer field. This field contains 0.

Table qualifier

A 128-byte character field. This field is not used for stored procedures.

Table name length

An unsigned 2-byte integer field. This field contains 0.

Table name

A 128-byte character field. This field is not used for stored procedures.

Column name length

An unsigned 2-byte integer field. This field contains 0.

Column name

A 128-byte character field. This field is not used for stored procedures.

Product information

An 8-byte character field that identifies the product on which the stored procedure executes. This field has the form *pppvrrm*, where:

- *ppp* is a 3-byte product code:
 - ARI** DB2 Server for VSE & VM
 - DSN** DB2 for z/OS
 - QSQ** DB2 for i
 - SQL** DB2 for Linux, UNIX, and Windows
- *vv* is a two-digit version identifier.
- *rr* is a two-digit release identifier.
- *m* is a one-digit maintenance level identifier.

Reserved area

2 bytes.

Operating system

A 4-byte integer field. It identifies the operating system on which the program that invokes the user-defined function 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	z/OS
13	Siemens Nixdorf
15	Windows 95
16	SCO UNIX
18	Linux
19	DYNIX/ptx
24	Linux for S/390
25	Linux for System z
26	Linux/IA64
27	Linux/PPC
28	Linux/PPC64
29	Linux/AMD64
400	iSeries

Number of entries in table function column list

An unsigned 2-byte integer field. This field contains 0.

Reserved area

26 bytes.

Table function column list pointer

This field is not used for stored procedures.

Unique application identifier

This field is a pointer to a string that uniquely identifies the application's connection to DB2. The string is regenerated at for each connection to DB2.

The string is the LUWID, which consists of a fully-qualified LU network name followed by a period and an LUW instance number. The LU network name consists of a one- to eight-character network ID, a period, and a one- to eight-character network LU name. The LUW instance number consists of 12 hexadecimal characters that uniquely identify the unit of work.

Reserved area

20 bytes.

Packages and plans for external stored procedures

An external stored procedure must have an associated package. The calling application can use either a plan or a package.

As part of the process of creating an external stored procedure, you prepare the procedure, which means that you precompile, compile, link-edit, and bind the application. The result of this process is a DB2 package. You do not need to create a DB2 plan for an external procedure. The procedure runs under the caller's thread and uses the plan from the client program that calls it.

The calling application can use a DB2 package or plan to execute the CALL statement.

Both the stored procedure package and the calling application plan or package must exist on the server before you run the calling application.

The following figure shows this relationship between a client program and a stored procedure. In the figure, the client program, which was bound into package A, issues a CALL statement to program B. Program B is an external stored procedure in a WLM address space. This external stored procedure was bound into package B.

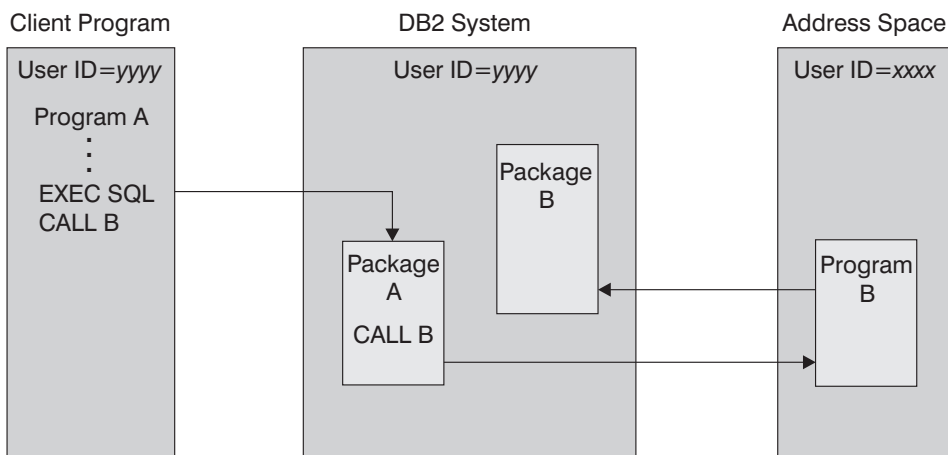


Figure 36. Stored procedure run time environment

You can control access to the stored procedure package by specifying the ENABLE bind option when you bind the package.

In the following situations, the stored procedure might use more than one package:

- You bind a DBRM several times into several versions of the same package, all of which have the same package name but reside in different package collections. Your stored procedure can switch from one version to another by using the SET CURRENT PACKAGESET statement.
- The stored procedure calls another program that contains SQL statements. This program has an associated package. This package must exist at the location where the stored procedure is defined and at the location where the SQL statements are executed.

Related reference:

- ➡ BIND and REBIND options for packages and plans (DB2 Commands)
- ➡ BIND PACKAGE (DSN) (DB2 Commands)
- ➡ SET CURRENT PACKAGESET (DB2 SQL)

Accessing other sites in an external procedure

External procedures can access tables at other DB2 locations.

Stored procedures can access tables at other DB2 locations by using three-part object names or CONNECT statements.

Related concepts:

“Accessing distributed data by using three-part table names” on page 871

Accessing non-DB2 resources in your stored procedure

Applications that run in a stored procedures address space can access any resources that are available to z/OS address spaces. For example, they can access VSAM files, flat files, APPC/MVS conversations, and IMS or CICS transactions.

Accessing these resources from a stored procedure can be useful if you want to update older applications. Suppose that you have existing applications that access non-DB2 resources, but you want to use newer DB2 applications to access the same data. You do not need to rewrite the application or migrate the data to DB2. Instead, you can use stored procedures to execute the existing program or access the non-DB2 data directly.

When a stored procedure runs, the stored procedure uses the Recoverable Resource Manager Services (RRS) for commitment control. When DB2 commits or rolls back work, DB2 coordinates all updates that are made to recoverable resources by other RRS compliant resource managers in the z/OS system.

To access non-DB2 resources in your stored procedure:

1. Consider serializing access to non-DB2 resources within your application. Not all non-DB2 resources can tolerate concurrent access by multiple TCBs in the same address space.
2. To access CICS, use one of the following methods:
 - Stored procedure DSNACICS
 - Message Queue Interface (MQI) for asynchronous execution of CICS transactions
 - External CICS interface (EXCI) for synchronous execution of CICS transactions
 - Advanced Program-to-Program Communication (APPC), using the Common Programming Interface Communications (CPI Communications) application programming interface

If your system is running a release of CICS that uses z/OS RRS, z/OS RRS controls commitment of all resources.

3. To access IMS DL/I data, use one of the following methods
 - Open Database Access interface (ODBA)
 - Stored procedures DSNAIMS and DSNAIMS2

If your system is not running a release of IMS that uses z/OS RRS, take one of the following actions:

- Use the CICS EXCI interface to run a CICS transaction synchronously. That CICS transaction can, in turn, access DL/I data.
- Invoke IMS transactions asynchronously using the MQI.
- Use APPC through the Common Programming Interface (CPI) Communications application programming interface.

- Determine which of the following authorization IDs you want to use to access the non-DB2 resources.

Table 100. Authorization IDs for accessing non-DB2 resources from a stored procedure

ID that you want to use to access the non-DB2 resources	SECURITY value to specify in the CREATE PROCEDURE statement
The authorization ID that is associated with the stored procedures address space	SECURITY DB2
The authorization ID under which the CALL statement is executed	SECURITY USER
The authorization ID under which the CREATE PROCEDURE statement is executed	SECURITY DEFINER

- Issue the CREATE PROCEDURE statement with the appropriate SECURITY option that you determined in the previous step.

When the stored procedure runs, DB2 establishes a RACF environment for accessing non-DB2 resources and uses the specified authorization ID to access protected z/OS resources.

Related tasks:

Chapter 14, “Calling a stored procedure from your application,” on page 791

[➤](#) Implementing RRS for stored procedures during installation (DB2 Installation and Migration)

[➤](#) Controlling stored procedure access to non-DB2 resources by using RACF (Managing Security)

Related reference:

[➤](#) DSNACICS stored procedure (DB2 Administration Guide)

[➤](#) DSNAIMS stored procedure (DB2 Administration Guide)

[➤](#) DSNAIMS2 stored procedure (DB2 Administration Guide)

[➤](#) CREATE PROCEDURE (SQL - external) (DB2 SQL)

[➤](#) APPC/MVS Configuration (Multiplatform APPC Configuration Guide)

Related information:

[➤](#) Accessing CICS and IMS (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

[➤](#) External CICS interface (EXCI) (CICS Transaction Server for z/OS)

Writing an external procedure to access IMS databases

IMS Open Database Access (ODBA) support lets a DB2 stored procedure connect to an IMS DBCTL or IMS DB/DC system and issue DL/I calls to access IMS databases.

ODBA support uses RRS for syncpoint control of DB2 and IMS resources. Therefore, stored procedures that use ODBA can run only in WLM-established stored procedures address spaces.

When you write a stored procedure that uses ODBA, follow the rules for writing an IMS application program that issues DL/I calls.

IMS work that is performed in a stored procedure is in the same commit scope as the stored procedure. As with any other stored procedure, the calling application commits work.

A stored procedure that uses ODBA must issue a DPSB PREP call to deallocate a PSB when all IMS work under that PSB is complete. The PREP keyword tells IMS to move inflight work to an indoubt state. When work is in the indoubt state, IMS does not require activation of syncpoint processing when the DPSB call is executed. IMS commits or backs out the work as part of RRS two-phase commit when the stored procedure caller executes COMMIT or ROLLBACK.

A sample COBOL stored procedure and client program demonstrate accessing IMS data using the ODBA interface. The stored procedure source code is in member DSN8EC1 and is prepared by job DSNTEJ61. The calling program source code is in member DSN8EC1 and is prepared and executed by job DSNTEJ62. All code is in data set DSNA10.SDSNSAMP.

The startup procedure for a stored procedures address space in which stored procedures that use ODBA run must include a DFSRESLB DD statement and an extra data set in the STEPLIB concatenation.

Related concepts:

➡ Installation step 19: Configure DB2 for running stored procedures and user-defined functions (DB2 Installation and Migration)

➡ Migration step 25: Configure DB2 for running stored procedures and user-defined functions (DB2 Installation and Migration)

Related information:

➡ Application programming design

Writing an external procedure to return result sets to a distributed client

An external procedure can return multiple query result sets to a distributed client if the value of DYNAMIC RESULT SETS in the stored procedure definition is greater than 0.

For each result set you want returned, your stored procedure must:

- Declare a cursor with the option WITH RETURN.
- Open the cursor.
- If the cursor is scrollable, ensure that the cursor is positioned before the first row of the result table.
- Leave the cursor open.

When the stored procedure ends, DB2 returns the rows in the query result set to the client.

DB2 does not return result sets for cursors that are closed before the stored procedure terminates. The stored procedure must execute a CLOSE statement for each cursor associated with a result set that should not be returned to the DRDA client.

Example: Declaring a cursor to return a result set: Suppose you want to return a result set that contains entries for all employees in department D11. First, declare a cursor that describes this subset of employees:

```
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
SELECT * FROM DSN8A10.EMP
WHERE WORKDEPT='D11';
```

Then, open the cursor:

```
EXEC SQL OPEN C1;
```

DB2 returns the result set and the name of the SQL cursor for the stored procedure to the client.

Use meaningful cursor names for returning result sets: The name of the cursor that is used to return result sets is made available to the client application through extensions to the DESCRIBE statement.

Use cursor names that are meaningful to the DRDA client application, especially when the stored procedure returns multiple result sets.

Objects from which you can return result sets: You can use any of these objects in the SELECT statement that is associated with the cursor for a result set:

- Tables, synonyms, views, created temporary tables, declared temporary tables, and aliases defined at the local DB2 subsystem

Returning a subset of rows to the client: If you execute FETCH statements with a result set cursor, DB2 does not return the fetched rows to the client program. For example, if you declare a cursor WITH RETURN and then execute the statements OPEN, FETCH, and FETCH, the client receives data beginning with the third row in the result set. If the result set cursor is scrollable and you fetch rows with it, you need to position the cursor before the first row of the result table after you fetch the rows and before the stored procedure ends.

Using a temporary table to return result sets: You can use a created temporary table or declared temporary table to return result sets from a stored procedure. This capability can be used to return nonrelational data to a DRDA client.

For example, you can access IMS data from a stored procedure in the following way:

- Use APPC/MVS to issue an IMS transaction.
- Receive the IMS reply message, which contains data that should be returned to the client.
- Insert the data from the reply message into a temporary table.
- Open a cursor against the temporary table. When the stored procedure ends, the rows from the temporary table are returned to the client.

Related tasks:

“Writing a program to receive the result sets from a stored procedure” on page 807

Restrictions when calling other programs from an external stored procedure

An external procedure can consist of more than one program, each with its own package. Your stored procedure can call other programs, stored procedures, or user-defined functions. Use the facilities of your programming language to call other programs.

If the stored procedure calls other programs that contain SQL statements, each of those called programs must have a DB2 package. The owner of the package or plan that contains the CALL statement must have EXECUTE authority for all packages that the other programs use.

When a stored procedure calls another program, DB2 determines which collection the package of the called program belongs to in one of the following ways:

- If the stored procedure definition contains PACKAGE PATH with a specified list of collection IDs, DB2 uses those collection IDs. If you also specify COLLID, DB2 ignores that clause.
- If the stored procedure definition contains COLLID *collection-id*, DB2 uses *collection-id*.
- If the stored procedure executes SET CURRENT PACKAGE PATH and contains the NO COLLID option, DB2 uses the CURRENT PACKAGE PATH special register. The package of the called program comes from the list of collections in the CURRENT PACKAGE PATH special register. For example, assume that CURRENT PACKAGE PATH contains the list COLL1, COLL2, COLL3, COLL4. DB2 searches for the first package (in the order of the list) that exists in these collections.
- If the stored procedure does not execute SET CURRENT PACKAGE PATH and instead executes SET CURRENT PACKAGESET, DB2 uses the CURRENT PACKAGESET special register. The package of the called program comes from the collection that is specified in the CURRENT PACKAGESET special register.
- If both of the following conditions are true, DB2 uses the collection ID of the package that contains the SQL statement CALL:
 - the stored procedure does not execute SET CURRENT PACKAGE PATH or SET CURRENT PACKAGESET
 - the stored procedure definition contains the NO COLLID option

When control returns from the stored procedure, the value of the CURRENT PACKAGESET special register is reset. DB2 restores the value of the CURRENT PACKAGESET special register to the value that it contained before the client program executed the SQL statement CALL.

Creating an external stored procedure as reentrant

Reentrant code is code for which a single copy can be used concurrently by two or more processes. For improved performance, prepare your stored procedures to be reentrant whenever possible

Reentrant stored procedures can improve performance for the following reasons:

- A reentrant stored procedure does not need to be loaded into storage every time that it is called.
- A single copy of the stored procedure can be shared by multiple tasks in the stored procedures address space. This sharing decreases the amount of virtual storage that is used for code in the stored procedures address space.

To create an external stored procedure as reentrant:

1. Compile the procedure as reentrant and link-edit it as reentrant and reusable.

For instructions on compiling programs to be reentrant, see the information for the programming language that you are using. For C and C++ procedures, you can use the z/OS binder to produce reentrant and reusable load modules.

If your stored procedure cannot be reentrant, link-edit it as non-reentrant and non-reusable. The non-reusable attribute prevents multiple tasks from using a single copy of the stored procedure at the same time.

2. Specify `STAY RESIDENT YES` in the `CREATE PROCEDURE` or `ALTER PROCEDURE` statement for the stored procedure. This option makes a reentrant stored procedure remain in storage.

A non-reentrant stored procedure must not remain in storage. You therefore need to specify `STAY RESIDENT NO` in the `CREATE PROCEDURE` or `ALTER PROCEDURE` statement for a non-reentrant stored procedure. `STAY RESIDENT NO` is the default.

Related concepts:

[➡ Making programs reentrant \(Enterprise COBOL for z/OS Programming Guide\)](#)

Related reference:

[➡ Compiler options \(COBOL\) \(Enterprise COBOL for z/OS Programming Guide\)](#)

[➡ ALTER PROCEDURE \(external\) \(DB2 SQL\)](#)

[➡ CREATE PROCEDURE \(external\) \(DB2 SQL\)](#)

[➡ Binder options reference \(MVS Program Management: User's Guide and Reference\)](#)

[➡ OPTIONS\(REENTRANT\) \(Enterprise PL/I for z/OS Compiler and Runtime Migration Guide\)](#)

[➡ Compile-time option descriptions \(PL/I\) \(Enterprise PL/I for z/OS Programming Guide:\)](#)

[➡ Reentrancy \(XL C/C++ User's Guide\)](#)

External stored procedures as main programs and subprograms

A stored procedure that runs in a WLM-established address space and uses Language Environment Release 1.7 or a subsequent release can be either a main program or a subprogram. A stored procedure that runs as a subprogram can perform better because Language Environment does less processing for it.

In general, a subprogram must do the following extra tasks that Language Environment performs for a main program:

- Initialization and cleanup processing
- Allocating and freeing storage
- Closing all open files before exiting

When you code stored procedures as subprograms, follow these rules:

- Follow the language rules for a subprogram. For example, you cannot perform I/O operations in a PL/I subprogram.
- Avoid using statements that terminate the Language Environment enclave when the program ends. Examples of such statements are `STOP` or `EXIT` in a PL/I subprogram, or `STOP RUN` in a COBOL subprogram. If the enclave terminates when a stored procedure ends, and the client program calls another stored procedure that runs as a subprogram, Language Environment must build a new enclave. As a result, the benefits of coding a stored procedure as a subprogram are lost.
- In COBOL stored procedures that are defined as `PROGRAM TYPE SUB` and `STAY RESIDENT YES`, if you use stored procedure parameters as host variables,

set the SQL-INIT-FLAG variable to 0. This variable is generated by the DB2 precompiler. Setting it to 0 ensures that the SQLDA is updated with the current addresses.

The following table summarizes the characteristics that define a main program and a subprogram.

Table 101. Characteristics of main programs and subprograms

Language	Main program	Subprogram
Assembler	MAIN=YES is specified in the invocation of the CEEENTRY macro.	MAIN=NO is specified in the invocation of the CEEENTRY macro.
C	Contains a main() function. Pass parameters to it through argc and argv.	A fetchable function. Pass parameters to it explicitly.
COBOL	A COBOL program that ends with GOBACK	A dynamically loaded subprogram that ends with GOBACK
PL/I	Contains a procedure declared with OPTIONS(MAIN)	A procedure declared with OPTIONS(FETCHABLE)

The following code shows an example of coding a C stored procedure as a subprogram.

```

/*****
/* This C subprogram is a stored procedure that uses linkage */
/* convention GENERAL and receives 3 parameters. */
/*****
#pragma linkage(cfunc,fetchable)
#include <stdlib.h>
void cfunc(char p1[11],long *p2,short *p3)
{
    /*****
    /* Declare variables used for SQL operations. These variables */
    /* are local to the subprogram and must be copied to and from */
    /* the parameter list for the stored procedure call. */
    /*****
EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
EXEC SQL END DECLARE SECTION;

    /*****
    /* Receive input parameter values into local variables. */
    /*****
strcpy(parm1,p1);
parm2 = *p2;
parm3 = *p3;
    /*****
    /* Perform operations on local variables. */
    /*****

:
    /*****
    /* Set values to be passed back to the caller. */
    /*****
strcpy(parm1,"SETBYSP");
parm2 = 100;
parm3 = 200;
    /*****
    /* Copy values to output parameters. */
    /*****

```



```

    strcpy(p1,parm1);
    *p2 = parm2;
    *p3 = parm3;
}

```

The following code shows an example of coding a C++ stored procedure as a subprogram.

```

/*****
/* This C++ subprogram is a stored procedure that uses linkage */
/* convention GENERAL and receives 3 parameters. */
/* The extern statement is required. */
/*****
extern "C" void cppfunc(char p1[11],long *p2,short *p3);
#pragma linkage(cppfunc,fetchable)
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;
void cppfunc(char p1[11],long *p2,short *p3)
{
    /*****
    /* Declare variables used for SQL operations. These variables */
    /* are local to the subprogram and must be copied to and from */
    /* the parameter list for the stored procedure call. */
    /*****
EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
EXEC SQL END DECLARE SECTION;

    /*****
    /* Receive input parameter values into local variables. */
    /*****
strcpy(parm1,p1);
parm2 = *p2;
parm3 = *p3;
    /*****
    /* Perform operations on local variables. */
    /*****

:
    /*****
    /* Set values to be passed back to the caller. */
    /*****
strcpy(parm1,"SETBYSP");
parm2 = 100;
parm3 = 200;
    /*****
    /* Copy values to output parameters. */
    /*****
strcpy(p1,parm1);
*p2 = parm2;
*p3 = parm3;
}

```

Data types in stored procedures

A stored procedure that is written in any language except REXX must declare each parameter that is passed to it. The definition for that stored procedure must also contain a compatible SQL data type declaration for each parameter.

For languages other than REXX

For all data types except LOBs, ROWIDs, locators, and VARCHARs (for C language), see the tables listed in the following table for the host data types that are compatible with the data types in the stored procedure definition. You cannot

l

have XML parameters in an external procedure.

For LOBs, ROWIDs, VARCHARs, and locators, the following table shows compatible declarations for the assembler language.

Table 102. Compatible assembler language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	Assembler declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	DS FL4
BLOB(<i>n</i>)	If <i>n</i> <= 65535: var DS 0FL4 var_length DS FL4 var_data DS CL <i>n</i> If <i>n</i> > 65535: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+(<i>n</i> -65535)
CLOB(<i>n</i>)	If <i>n</i> <= 65535: var DS 0FL4 var_length DS FL4 var_data DS CL <i>n</i> If <i>n</i> > 65535: var DS 0FL4 var_length DS FL4 var_data DS CL65535 ORG var_data+(<i>n</i> -65535)
DBCLOB(<i>n</i>)	If <i>m</i> (=2* <i>n</i>) <= 65534: var DS 0FL4 var_length DS FL4 var_data DS CL <i>m</i> If <i>m</i> > 65534: var DS 0FL4 var_length DS FL4 var_data DS CL65534 ORG var_data+(<i>m</i> -65534)
ROWID	DS HL2,CL40
VARCHAR(<i>n</i>)	If PARAMETER VARCHAR NULTERM is specified or implied: char data[<i>n</i> +1]; If PARAMETER VARCHAR STRUCTURE is specified: struct {short len; char data[<i>n</i>]; } var;

Note:

1. This row does not apply to VARCHAR(*n*) FOR BIT DATA. BIT DATA is always passed in a structured representation.

For LOBs, ROWIDs, and locators, the following table shows compatible declarations for the C language.

Table 103. Compatible C language declarations for LOBs, ROWIDs, and locators

SQL data type in definition	C declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	unsigned long
BLOB(<i>n</i>)	struct {unsigned long length; char data[<i>n</i>]; } var;
CLOB(<i>n</i>)	struct {unsigned long length; char var_data[<i>n</i>]; } var;
DBCLOB(<i>n</i>)	struct {unsigned long length; sqldbchar data[<i>n</i>]; } var;
ROWID	struct {short int length; char data[40]; } var;

For LOBs, ROWIDs, and locators, the following table shows compatible declarations for COBOL.

Table 104. Compatible COBOL declarations for LOBs, ROWIDs, and locators

SQL data type in definition	COBOL declaration
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	01 var PIC S9(9) COMP-5.
BLOB(<i>n</i>)	01 var. 49 var-LENGTH PIC S9(9) COMP-5. 49 var-DATA PIC X(<i>n</i>).
CLOB(<i>n</i>)	01 var. 49 var-LENGTH PIC S9(9) COMP-5. 49 var-DATA PIC X(<i>n</i>).
DBCLOB(<i>n</i>)	01 var. 49 var-LENGTH PIC S9(9) COMP-5. 49 var-DATA PIC G(<i>n</i>) DISPLAY-1.
ROWID	01 var. 49 var-LEN PIC S9(4) COMP-5. 49 var-DATA PIC X(40).

For LOBs, ROWIDs, and locators, the following table shows compatible declarations for PL/I.

Table 105. Compatible PL/I declarations for LOBs, ROWIDs, and locators

SQL data type in definition	PL/I
TABLE LOCATOR BLOB LOCATOR CLOB LOCATOR DBCLOB LOCATOR	BIN FIXED(31)
BLOB(<i>n</i>)	<pre> If n <= 32767: 01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA CHAR(<i>n</i>); If n > 32767: 01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(<i>n</i>) CHAR(32767), 03 var_DATA2 CHAR(mod(<i>n</i>,32767)); </pre>
CLOB(<i>n</i>)	<pre> If n <= 32767: 01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA CHAR(<i>n</i>); If n > 32767: 01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(<i>n</i>) CHAR(32767), 03 var_DATA2 CHAR(mod(<i>n</i>,32767)); </pre>
DBCLOB(<i>n</i>)	<pre> If n <= 16383: 01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA GRAPHIC(<i>n</i>); If n > 16383: 01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(<i>n</i>) GRAPHIC(16383), 03 var_DATA2 GRAPHIC(mod(<i>n</i>,16383)); </pre>
ROWID	CHAR(40) VAR

Tables of results: Each high-level language definition for stored procedure parameters supports only a single instance (a scalar value) of the parameter. There

is no support for structure, array, or vector parameters. Because of this, the SQL statement CALL limits the ability of an application to return some kinds of tables. For example, an application might need to return a table that represents multiple occurrences of one or more of the parameters passed to the stored procedure. Because the SQL statement CALL cannot return more than one set of parameters, use one of the following techniques to return such a table:

- Put the data that the application returns in a DB2 table. The calling program can receive the data in one of these ways:
 - The calling program can fetch the rows from the table directly. Specify FOR FETCH ONLY or FOR READ ONLY on the SELECT statement that retrieves data from the table. A block fetch can retrieve the required data efficiently.
 - The stored procedure can return the contents of the table as a result set. See “Writing an external procedure to return result sets to a distributed client” on page 641 and “Writing a program to receive the result sets from a stored procedure” on page 807 for more information.
- Convert tabular data to string format and return it as a character string parameter to the calling program. The calling program and the stored procedure can establish a convention for interpreting the content of the character string. For example, the SQL statement CALL can pass a 1920-byte character string parameter to a stored procedure, which enables the stored procedure to return a 24x80 screen image to the calling program.

Related concepts:

“Compatibility of SQL and language data types” on page 180

 Installation step 19: Configure DB2 for running stored procedures and user-defined functions (DB2 Installation and Migration)

 Migration step 25: Configure DB2 for running stored procedures and user-defined functions (DB2 Installation and Migration)

REXX stored procedures

A REXX stored procedure is similar to any other REXX procedure and follows the same rules as stored procedures in other languages. A REXX stored procedure receives input parameters, executes REXX commands, optionally executes SQL statements, and returns at most one output parameter. However, a few differences exist.

A REXX stored procedure is different from other REXX procedures in the following ways:

- A REXX stored procedure must not execute any of the following DSNREXX commands that are used for the DB2 subsystem thread attachment:

```
ADDRESS DSNREXX CONNECT
ADDRESS DSNREXX DISCONNECT
CALL SQLDBS ATTACH TO
CALL SQLDBS DETACH
```

When you execute SQL statements in your stored procedure, DB2 establishes the connection for you.

- A REXX stored procedure must run in a WLM-established stored procedures address space.
- A language REXX stored procedure executes in a background TSO/E REXX environment provided by the TSO/E environment service IKJTSOEV.

Unlike other stored procedures, you do not prepare REXX stored procedures for execution. REXX stored procedures run using one of four packages that are bound during the installation of DB2 REXX Language Support. The current isolation level at which the stored procedure runs depends on the package that DB2 uses when the stored procedure runs:

Package name	Isolation level
DSNREXRR	Repeatable read (RR)
DSNREXRS	Read stability (RS)
DSNREXCS	Cursor stability (CS)
DSNREXUR	Uncommitted read (UR)

This topic shows an example of a REXX stored procedure that executes DB2 commands. The stored procedure performs the following actions:

- Receives one input parameter, which contains a DB2 command.
- Calls the IFI COMMAND function to execute the command.
- Extracts the command result messages from the IFI return area and places the messages in a created temporary table. Each row of the temporary table contains a sequence number and the text of one message.
- Opens a cursor to return a result set that contains the command result messages.
- Returns the unformatted contents of the IFI return area in an output parameter.

The following example shows the definition of the stored procedure.

```
CREATE PROCEDURE COMMAND(IN CMDTEXT VARCHAR(254), OUT CMDRESULT VARCHAR(32704))
  LANGUAGE REXX
  EXTERNAL NAME COMMAND
  NO COLLID
  ASUTIME NO LIMIT
  PARAMETER STYLE GENERAL
  STAY RESIDENT NO
  RUN OPTIONS 'TRAP(ON)'
  WLM ENVIRONMENT WLMENV1
  SECURITY DB2
  DYNAMIC RESULT SETS 1
  COMMIT ON RETURN NO;
```

The following example shows the COMMAND stored procedure that executes DB2 commands.

```
/* REXX */
PARSE UPPER ARG CMD                                /* Get the DB2 command text */
                                                    /* Remove enclosing quotation marks */
IF LEFT(CMD,2) = '"' & RIGHT(CMD,2) = '"' THEN
CMD = SUBSTR(CMD,2,LENGTH(CMD)-2)
ELSE
IF LEFT(CMD,2) = '''' & RIGHT(CMD,2) = '''' THEN
CMD = SUBSTR(CMD,3,LENGTH(CMD)-4)
COMMAND = SUBSTR("COMMAND",1,18," ")
/*****
/* Set up the IFCA, return area, and output area for the */
/* IFI COMMAND call. */
*****/
IFCA = SUBSTR('00'X,1,180,'00'X)
```

```

IFCA = OVERLAY(D2C(LENGTH(IFCA),2),IFCA,1+0)
IFCA = OVERLAY("IFCA",IFCA,4+1)
RTRNAREASIZE = 262144 /*1048572*/
RTRNAREA = D2C(RTRNAREASIZE+4,4)LEFT(' ',RTRNAREASIZE,' ')
OUTPUT = D2C(LENGTH(CMD)+4,2)||'0000'X||CMD
BUFFER = SUBSTR(" ",1,16," ")
/*****
/* Make the IFI COMMAND call. */
*****/
ADDRESS LINKPGM "DSNLIR COMMAND IFCA RTRNAREA OUTPUT"
WRC = RC
RTRN= SUBSTR(IFCA,12+1,4)
REAS= SUBSTR(IFCA,16+1,4)
TOTLEN = C2D(SUBSTR(IFCA,20+1,4))
/*****
/* Set up the host command environment for SQL calls. */
*****/
"SUBCOM DSNREXX" /* Host cmd env available? */
IF RC THEN /* No--add host cmd env */
  S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
/*****
/* Set up SQL statements to insert command output messages */
/* into a temporary table. */
*****/
SQLSTMT='INSERT INTO SYSIBM.SYSPRINT(SEQNO,TEXT) VALUES(?,?)'
ADDRESS DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1"
IF SQLCODE = 0 THEN CALL SQLCA
ADDRESS DSNREXX "EXECSQL PREPARE S1 FROM :SQLSTMT"
IF SQLCODE = 0 THEN CALL SQLCA
/*****
/* Extract messages from the return area and insert them into */
/* the temporary table. */
*****/
SEQNO = 0
OFFSET = 4+1
DO WHILE ( OFFSET < TOTLEN )
  LEN = C2D(SUBSTR(RTRNAREA,OFFSET,2))
  SEQNO = SEQNO + 1
  TEXT = SUBSTR(RTRNAREA,OFFSET+4,LEN-4-1)
  ADDRESS DSNREXX "EXECSQL EXECUTE S1 USING :SEQNO,:TEXT"
  IF SQLCODE = 0 THEN CALL SQLCA
  OFFSET = OFFSET + LEN
END
/*****
/* Set up a cursor for a result set that contains the command */
/* output messages from the temporary table. */
*****/
SQLSTMT='SELECT SEQNO,TEXT FROM SYSIBM.SYSPRINT ORDER BY SEQNO'
ADDRESS DSNREXX "EXECSQL DECLARE C2 CURSOR FOR S2"
IF SQLCODE = 0 THEN CALL SQLCA
ADDRESS DSNREXX "EXECSQL PREPARE S2 FROM :SQLSTMT"
IF SQLCODE = 0 THEN CALL SQLCA
/*****
/* Open the cursor to return the message output result set to */
/* the caller. */
*****/
ADDRESS DSNREXX "EXECSQL OPEN C2"
IF SQLCODE = 0 THEN CALL SQLCA
S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX') /* REMOVE CMD ENV */
EXIT SUBSTR(RTRNAREA,1,TOTLEN+4)
/*****
/* Routine to display the SQLCA */
*****/
SQLCA:
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC

```

```

SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD ='SQLERRD.1',' ,
      | SQLERRD.2',' ,
      | SQLERRD.3',' ,
      | SQLERRD.4',' ,
      | SQLERRD.5',' ,
      | SQLERRD.6
SAY 'SQLWARN ='SQLWARN.0',' ,
      | SQLWARN.1',' ,
      | SQLWARN.2',' ,
      | SQLWARN.3',' ,
      | SQLWARN.4',' ,
      | SQLWARN.5',' ,
      | SQLWARN.6',' ,
      | SQLWARN.7',' ,
      | SQLWARN.8',' ,
      | SQLWARN.9',' ,
      | SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
SAY 'SQLCODE ='SQLCODE
EXIT 'SQLERRMC ='SQLERRMC';' ,
    | 'SQLERRP ='SQLERRP';' ,
    | 'SQLERRD ='SQLERRD.1',' ,
      | SQLERRD.2',' ,
      | SQLERRD.3',' ,
      | SQLERRD.4',' ,
      | SQLERRD.5',' ,
      | SQLERRD.6';' ,
    | 'SQLWARN ='SQLWARN.0',' ,
      | SQLWARN.1',' ,
      | SQLWARN.2',' ,
      | SQLWARN.3',' ,
      | SQLWARN.4',' ,
      | SQLWARN.5',' ,
      | SQLWARN.6',' ,
      | SQLWARN.7',' ,
      | SQLWARN.8',' ,
      | SQLWARN.9',' ,
      | SQLWARN.10';' ,
    | 'SQLSTATE='SQLSTATE';'

```

Related reference:

“Calling a stored procedure from a REXX procedure” on page 797

 TSO/E services available under IKJTSOEV (TSO/E Programming Services)

Modifying an external stored procedure definition

You can modify the definition of an external stored procedure or the stored procedure source code. In either case, you need to prepare the stored procedure again.

To modify an external stored procedure definition:

1. Issue the ALTER PROCEDURE statement with the appropriate options. This new definition replaces the existing definition.
2. Prepare the external stored procedure again, as you did when you originally created the external stored procedure.

Suppose that an existing C stored procedure was defined with the following statement:

```

CREATE PROCEDURE B(IN V1 INTEGER, OUT V2 CHAR(9))
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  EXTERNAL NAME SUMMOD

```

```

COLLID SUMCOLL
ASUTIME LIMIT 900
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON) '
WLM ENVIRONMENT PAYROLL
PROGRAM TYPE MAIN
SECURITY DB2
DYNAMIC RESULT SETS 10
COMMIT ON RETURN NO;

```

Assume that you need to make the following changes to the stored procedure definition:

- The stored procedure selects data from DB2 tables but does not modify DB2 data.
- The parameters can have null values, and the stored procedure can return a diagnostic string.
- The length of time that the stored procedure runs is unlimited.
- If the stored procedure is called by another stored procedure or a user-defined function, the stored procedure uses the WLM environment of the caller.

The following ALTER PROCEDURE statement makes these changes:

```

ALTER PROCEDURE B
  READS SQL DATA
  ASUTIME NO LIMIT
  PARAMETER STYLE SQL
  WLM ENVIRONMENT (PAYROLL,*);

```

Related reference:

 ALTER PROCEDURE (external) (DB2 SQL)

Creating multiple versions of external procedures and external SQL procedures

For native SQL procedures, you can use DB2 to create and maintain multiple versions of the procedure. For external SQL procedures and other external procedures, you must manually maintain multiple versions of the procedures.

To create multiple versions of external procedures and external SQL procedures, use one of the following techniques:

- Define multiple procedures with the same name in different schemas. You can subsequently use the SQL path to determine which version of the procedure is to be used by a calling program.
- Define multiple versions of the executable code. You can subsequently use a particular version by specifying the name of the load module for the version that you want to use on the EXTERNAL clause of the CREATE PROCEDURE statement or ALTER PROCEDURE statement.
- Define multiple packages for a procedure. You can subsequently use the COLLID option, the CURRENT PACKAGESET special register, or the CURRENT PACKAGE PATH special register to specify which version of the procedure is to be used by the calling application.
- Set up multiple WLM environments to use different versions of a procedure.

Chapter 11. Adding and modifying data

Your application program can add, modify, or delete data in any DB2 table for which you have the appropriate access.

Inserting data into tables

You can use several different methods to insert data into a table. Decide which method to use based on the amount of data that you need to insert and the other operations that your program needs to perform.

Besides using stand-alone INSERT statements, you can use the following ways to insert data into a table:

- You can use the MERGE statement to insert new data and update existing data in the same operation. .
- You can write an application program to prompt for and enter large amounts of data into a table.
- You can also use the DB2 LOAD utility to enter data from other sources.

Related tasks:

“Inserting data and updating data in a single operation” on page 661

Related reference:

 [LOAD \(DB2 Utilities\)](#)

Inserting rows by using the INSERT statement

One way to insert data into tables is to use the SQL INSERT statement. This method is useful for inserting small amounts of data or inserting data from another table or view.

Use an INSERT statement to add new rows to a table or view. Using an INSERT statement, you can do the following actions:

- Specify the column values to insert a single row. You can specify constants, host variables, expressions, DEFAULT, or NULL by using the VALUES clause.
- In an application program, specify arrays of column values to insert multiple rows into a table. Use host variable arrays in the VALUES clause of the INSERT FOR *n* ROWS statement to add multiple rows of column values to a table.
- Include a SELECT statement in the INSERT statement to tell DB2 that another table or view contains the data for the new row or rows.

In each case, for every row that you insert, you must provide a value for any column that does not have a default value. For a column that meets one of the following conditions, specify DEFAULT to tell DB2 to insert the default value for that column:

- The column is nullable.
- The column is defined with a default value.
- The column has data type ROWID. ROWID columns always have default values.
- The column is an identity column. Identity columns always have default values.
- The column is a row change timestamp column.

The values that you can insert into a ROWID column, an identity column, or a row change timestamp column depend on whether the column is defined with GENERATED ALWAYS or GENERATED BY DEFAULT.

Inserting a single row:

You can use the VALUES clause of the INSERT statement to insert a single row of column values into a table. You can either name all of the columns for which you are providing values, or you can omit the list of column names. If you omit the column name list, you must specify values for **all** of the columns.

Recommendation: For static INSERT statements, name all of the columns for which you are providing values for the following reasons:

- Your INSERT statement is independent of the table format. (For example, you do not need to change the statement when a column is added to the table.)
- You can verify that you are specifying the values in order.
- Your source statements are more self-descriptive.

If you do not name the columns in a static INSERT statement, and a column is added to the table, an error can occur if the INSERT statement is rebound. An error will occur after any rebind of the INSERT statement unless you change the INSERT statement to include a value for the new column. This is true even if the new column has a default value.

When you list the column names, you must specify their corresponding values in the same order as in the list of column names.

Example: The following statement inserts information about a new department into the YDEPT table.

```
INSERT INTO YDEPT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
VALUES ('E31', 'DOCUMENTATION', '000010', 'E01', '');
```

After inserting a new department row into your YDEPT table, you can use a SELECT statement to see what you have loaded into the table. The following SQL statement shows you all of the new department rows that you have inserted:

```
SELECT *
FROM YDEPT
WHERE DEPTNO LIKE 'E%'
ORDER BY DEPTNO;
```

The result table looks similar to the following output:

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
E01	SUPPORT SERVICES	000050	A00	-----
E11	OPERATIONS	000090	E01	-----
E21	SOFTWARE SUPPORT	000100	E01	-----
E31	DOCUMENTATION	000010	E01	-----

Example: The following statement inserts information about a new employee into the YEMP table. Because the WORKDEPT column is a foreign key, the value that is inserted for that column (E31) must be a value in the primary key column, which is DEPTNO in the YDEPT table.

```
INSERT INTO YEMP
VALUES ('000400', 'RUTHERFORD', 'B', 'HAYES', 'E31', '5678', '1998-01-01',
'MANAGER', 16, 'M', '1970-07-10', 24000, 500, 1900);
```

Example: The following statement also inserts a row into the YEMP table. Because the unspecified columns allow null values, DB2 inserts null values into the columns that you do not specify.

```
INSERT INTO YEMP
  (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, JOB)
VALUES ('000410', 'MILLARD', 'K', 'FILLMORE', 'D11', '4888', 'MANAGER');
```

Related concepts:

“Rules for inserting data into an identity column” on page 658

“Rules for inserting data into a ROWID column” on page 658

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

“Inserting rows into a table from another table”

Related reference:

 CREATE TABLE (DB2 SQL)

Inserting rows into a table from another table

You can copy one or more rows from one table into another table.

Use a fullselect within an INSERT statement to select rows from one table to insert into another table.

Example: The following SQL statement creates a table named TELE:

```
CREATE TABLE TELE
  (NAME2 VARCHAR(15) NOT NULL,
   NAME1 VARCHAR(12) NOT NULL,
   PHONE CHAR(4));
```

The following statement copies data from DSN8A10.EMP into the newly created table:

```
INSERT INTO TELE
  SELECT LASTNAME, FIRSTNME, PHONENO
  FROM DSN8A10.EMP
  WHERE WORKDEPT = 'D21';
```

The two previous statements create and fill a table, TELE, that looks similar to the following table:

NAME2	NAME1	PHONE
=====	=====	=====
PULASKI	EVA	7831
JEFFERSON	JAMES	2094
MARINO	SALVATORE	3780
SMITH	DANIEL	0961
JOHNSON	SYBIL	8953
PEREZ	MARIA	9001
MONTEVERDE	ROBERT	3780

The CREATE TABLE statement example creates a table which, at first, is empty. The table has columns for last names, first names, and phone numbers, but does not have any rows.

The INSERT statement fills the newly created table with data that is selected from the DSN8A10.EMP table: the names and phone numbers of employees in department D21.

Example: The following CREATE statement creates a table that contains an employee's department name and phone number. The fullselect within the INSERT

statement fills the DLIST table with data from rows that are selected from two existing tables, DSN8A10.DEPT and DSN8A10.EMP.

```
CREATE TABLE DLIST
  (DEPT   CHAR(3)      NOT NULL,
   DNAME  VARCHAR(36)  ,
   LNAME  VARCHAR(15)  NOT NULL,
   FNAME  VARCHAR(12)  NOT NULL,
   INIT   CHAR         ,
   PHONE  CHAR(4) );

INSERT INTO DLIST
  SELECT DEPTNO, DEPTNAME, LASTNAME, FIRSTNAME, MIDINIT, PHONENO
  FROM DSN8A10.DEPT, DSN8A10.EMP
  WHERE DEPTNO = WORKDEPT;
```

Rules for inserting data into a ROWID column

A *ROWID column* contains unique values that identify each row in a table. Whether you can insert data into a ROWID column and how that data gets inserted depends on how the column is defined.

A *ROWID column* is a column that is defined with a ROWID data type. You must have a column with a ROWID data type in a table that contains a LOB column. The ROWID column is stored in the base table and is used to look up the actual LOB data in the LOB table space. In addition, a ROWID column enables you to write queries that navigate directly to a row in a table. For information about using ROWID columns for direct-row access, see “Specifying direct row access by using row IDs” on page 754.

Before you insert data into a ROWID column, you must know how the ROWID column is defined. ROWID columns can be defined as GENERATED ALWAYS or GENERATED BY DEFAULT. GENERATED ALWAYS means that DB2 generates a value for the column, and you cannot insert data into that column. If the column is defined as GENERATED BY DEFAULT, you can insert a value, and DB2 provides a default value if you do not supply one.

Example: Suppose that tables T1 and T2 have two columns: an integer column and a ROWID column. For the following statement to run successfully, ROWIDCOL2 must be defined as GENERATED BY DEFAULT.

```
INSERT INTO T2 (INTCOL2,ROWIDCOL2)
  SELECT * FROM T1;
```

If ROWIDCOL2 is defined as GENERATED ALWAYS, you cannot insert the ROWID column data from T1 into T2, but you can insert the integer column data. To insert only the integer data, use one of the following methods:

- Specify only the integer column in your INSERT statement, as in the following statement:

```
INSERT INTO T2 (INTCOL2)
  SELECT INTCOL1 FROM T1;
```

- Specify the OVERRIDING USER VALUE clause in your INSERT statement to tell DB2 to ignore any values that you supply for system-generated columns, as in the following statement:

```
INSERT INTO T2 (INTCOL2,ROWIDCOL2) OVERRIDING USER VALUE
  SELECT * FROM T1;
```

Rules for inserting data into an identity column

An *identity column* contains a unique numeric value for each row in the table. Whether you can insert data into an identity column and how that data gets inserted depends on how the column is defined.

An *identity column* is a numeric column, defined in a CREATE TABLE or ALTER TABLE statement, that has ascending or descending values. For an identity column to be as useful as possible, its values should also be unique. The column has a SMALLINT, INTEGER, or DECIMAL(*p*,0) data type and is defined with the AS IDENTITY clause. The AS IDENTITY clause specifies that the column is an identity column. For information about using identity columns to uniquely identify rows, see “Identity columns” on page 466

Before you insert data into an identity column, you must know how the column is defined. Identity columns are defined with the GENERATED ALWAYS or GENERATED BY DEFAULT clause. GENERATED ALWAYS means that DB2 generates a value for the column, and you cannot insert data into that column. If the column is defined as GENERATED BY DEFAULT, you can insert a value, and DB2 provides a default value if you do not supply one.

Example: Suppose that tables T1 and T2 have two columns: a character column and an integer column that is defined as an identity column. For the following statement to run successfully, IDENTCOL2 must be defined as GENERATED BY DEFAULT.

```
INSERT INTO T2 (CHARCOL2,IDENTCOL2)
  SELECT * FROM T1;
```

If IDENTCOL2 is defined as GENERATED ALWAYS, you cannot insert the identity column data from T1 into T2, but you can insert the character column data. To insert only the character data, use one of the following methods:

- Specify only the character column in your INSERT statement, as in the following statement:

```
INSERT INTO T2 (CHARCOL2)
  SELECT CHARCOL1 FROM T1;
```

- Specify the OVERRIDING USER VALUE clause in your INSERT statement to tell DB2 to ignore any values that you supply for system-generated columns, as in the following statement:

```
INSERT INTO T2 (CHARCOL2,IDENTCOL2) OVERRIDING USER VALUE
  SELECT * FROM T1;
```

Restrictions when assigning values to columns with distinct types

Certain conditions are required when you assign a column value to another column or when you assign a constant to a column of a distinct type. If the conditions are not met, you cannot assign the value.

When assigning a column value to another column or a constant to a column of a distinct type, the type of the value that is to be assigned must match the column type, or you must be able to cast one type to the other. Otherwise, you cannot assign the value.

If you need to assign a value of one distinct type to a column of another distinct type, a function must exist that converts the value from one type to another. Because DB2 provides cast functions only between distinct types and their source types, you must write the function to convert from one distinct type to another.

Assigning column values to columns with different distinct types

Suppose tables JAPAN_SALES and JAPAN_SALES_03 are defined like this:

```

CREATE TABLE JAPAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1990),
   TOTAL        JAPANESE_YEN);

CREATE TABLE JAPAN_SALES_03
  (PRODUCT_ITEM  INTEGER,
   TOTAL        US_DOLLAR);

```

You need to insert values from the TOTAL column in JAPAN_SALES into the TOTAL column of JAPAN_SALES_03. Because INSERT statements follow assignment rules, DB2 does not let you insert the values directly from one column to the other because the columns are of different distinct types. Suppose that a user-defined function called US_DOLLAR has been written that accepts values of type JAPANESE_YEN as input and returns values of type US_DOLLAR. You can then use this function to insert values into the JAPAN_SALES_03 table:

```

INSERT INTO JAPAN_SALES_03
  SELECT PRODUCT_ITEM, US_DOLLAR(TOTAL)
  FROM JAPAN_SALES
  WHERE YEAR = 2003;

```

Assigning column values with distinct types to host variables

The rules for assigning distinct types to host variables or host variables to columns of distinct types differ from the rules for constants and columns.

You can assign a column value of a distinct type to a host variable if you can assign a column value of the distinct type's source type to the host variable. In the following example, you can assign SIZECOL1 and SIZECOL2, which has distinct type SIZE, to host variables of type double and short because the source type of SIZE, which is INTEGER, can be assigned to host variables of type double or short.

```

EXEC SQL BEGIN DECLARE SECTION;
  double hv1;
  short  hv2;
EXEC SQL END DECLARE SECTION;
CREATE DISTINCT TYPE SIZE AS INTEGER;
CREATE TABLE TABLE1 (SIZECOL1 SIZE, SIZECOL2 SIZE);
:
:
SELECT SIZECOL1, SIZECOL2
  INTO :hv1, :hv2
  FROM TABLE1;

```

Assigning host variable values to columns with distinct types

When you assign a value in a host variable to a column with a distinct type, the type of the host variable must be able to cast to the distinct type.

In this example, values of host variable hv2 can be assigned to columns SIZECOL1 and SIZECOL2, because C data type short is equivalent to DB2 data type SMALLINT, and SMALLINT is promotable to data type INTEGER. However, values of hv1 cannot be assigned to SIZECOL1 and SIZECOL2, because C data type double, which is equivalent to DB2 data type DOUBLE, is not promotable to data type INTEGER.

```

EXEC SQL BEGIN DECLARE SECTION;
  double hv1;
  short  hv2;
EXEC SQL END DECLARE SECTION;
CREATE DISTINCT TYPE SIZE AS INTEGER;
CREATE TABLE TABLE1 (SIZECOL1 SIZE, SIZECOL2 SIZE);

```

```

:
INSERT INTO TABLE1
  VALUES (:hv1,:hv1);      /* Invalid statement */
INSERT INTO TABLE1
  VALUES (:hv2,:hv2);      /* Valid statement  */

```

Related concepts:

 Promotion of data types (DB2 SQL)

Inserting data and updating data in a single operation

You can update existing data and insert new data in a single operation. This operation is useful when you want to update a table with a set of rows, some of which are changes to existing rows and some of which are new rows.

You can update existing data and insert new data in a single operation by using the MERGE statement.

For example, an application might request a set of rows from a database, enable a user to modify the data through a GUI, and then store the modified data in the database. Some of this modified data is updates to existing rows, and some of this data is new rows. You can do these update and insert operations in one step.

To update existing data and inserting new data, specify a MERGE statement with the WHEN MATCHED and WHEN NOT MATCHED clauses. These clauses specify how DB2 handles matched and unmatched data. If DB2 finds a matching row, that row is updated. If DB2 does not find a matching row, a new row is inserted.

Example: Suppose that you need to update the inventory at a car dealership. You need to add new car models to the inventory and update information about car models that are already in the inventory.

You could make these changes with the following series of statements:

```

UPDATE INVENTORY
  SET QUANTITY = QUANTITY + :hv_delta
  WHERE MODEL = :hv_model;

--begin pseudo code
if sqlcode >= 0
then do
  GD
  if rc = 0 then INSERT..
end
-- end pseudo code

GET DIAGNOSTICS :rc = ROW_COUNT;

IF rc = 0 THEN
INSERT INTO INVENTORY VALUES (:hv_model, :hv_delta);
END IF;

```

The MERGE statement simplifies the update and the insert into a single statement:

```

MERGE INTO INVENTORY
USING ( VALUES (:hv_model, :hv_delta) ) AS SOURCE(MODEL, DELTA)
ON INVENTORY.MODEL = SOURCE.MODEL
  WHEN MATCHED THEN UPDATE SET QUANTITY = QUANTITY + SOURCE.DELTA
  WHEN NOT MATCHED THEN INSERT VALUES (SOURCE.MODEL, SOURCE.DELTA)
NOT ATOMIC CONTINUE ON SQLEXCEPTION;

```


Selecting values while merging data

When you update existing data and insert new data in a single merge operation, you can select values from those rows at the same time.

You can select values from rows that are being merged by specifying the MERGE statement in the FROM clause of the SELECT statement. When you merge one or more rows into a table, you can retrieve:

- The value of an automatically generated column such as a ROWID or identity column
- Any default values for columns
- All values for a merged row, without specifying individual column names
- Calculated values based on the changes to merged rows

Specify the FINAL TABLE clause with SELECT FROM MERGE statements. The FINAL TABLE consists of the rows of the table or view after the merge occurs.

Example: Suppose that you need to input data into the STOCK table, which contains company stock symbols and stock prices from your stock portfolio. Some of your input data refers to companies that are already in the STOCK table; some of the data refers to companies that you are adding to your stock portfolio. If the stock symbol exists in the SYMBOL column of the STOCK table, you need to update the PRICE column. If the company stock symbol is not yet in the STOCK table, you need to insert a new row with the stock symbol and the stock price. Furthermore, you need to add a new value DELTA to your output to show the change in stock price.

Suppose that the STOCK table contains the data that is shown in Table 106.

Table 106. STOCK table before SELECT FROM MERGE statement

SYMBOL	PRICE
XCOM	95.00
YCOM	24.50

Now, suppose that :hv_symbol and :hv_price are host variable arrays that contain updated data that corresponds to the data that is shown in Table 106. Table 107 shows the host variable data for stock activity.

Table 107. Host variable arrays of stock activity

hv_symbol	hv_price
XCOM	97.00
NEWC	30.00
XCOM	107.00

NEWC is new to the STOCK table, so its symbol and price need to be inserted into the STOCK table. The rows for XCOM in Table 107 represent changed stock prices, so these values need to be updated in the STOCK table. Also, the output needs to show the change in stock prices as a DELTA value.

The following SELECT FROM MERGE statement updates the price of XCOM, inserts the symbol and price for NEWC, and returns an output that includes a DELTA value for the change in stock price.


```

SELECT SYMBOL, PRICE, DELTA FROM FINAL TABLE
(MERGE INTO STOCK AS S INCLUDE (DELTA DECIMAL(5,20)
USING (:hv_symbol, :hv_price) FOR :hv_nrows ROWS) AS R (SYMBOL, PRICE)
ON S.SYMBOL = R.SYMBOL
WHEN MATCHED THEN UPDATE SET
DELTA = R.PRICE - S.PRICE, PRICE=R.PRICE
WHEN NOT MATCHED THEN INSERT
(SYMBOL, PRICE, DELTA) VALUES (R.SYMBOL, R.PRICE, R.PRICE)
NOT ATOMIC CONTINUE ON SQLEXCEPTION);

```

The INCLUDE clause specifies that an additional column, DELTA, can be returned in the output without adding a column to the STOCK table. The UPDATE portion of the MERGE statement sets the DELTA value to the differential of the previous stock price with the value set for the update operation. The INSERT portion of the MERGE statement sets the DELTA value to the same value as the PRICE column.

After the SELECT FROM MERGE statement is processed, the STOCK table contains the data that is shown in Table 108.

Table 108. STOCK table after SELECT FROM MERGE statement

SYMBOL	PRICE
XCOM	107.00
YCOM	24.50
NEWC	30.00

The following output of the SELECT FROM MERGE statement includes both updates to XCOM and a DELTA value for each output row.

```

SYMBOL    PRICE    DELTA
-----
XCOM      97.00    2.00
NEWC      30.00    30.00
XCOM      107.00   10.00

```

Selecting values while inserting data

When you insert rows into a table, you can select values from those rows at the same time.

You can select values from rows that are being inserted by specifying the INSERT statement in the FROM clause of the SELECT statement. When you insert one or more new rows into a table, you can retrieve:

- The value of an automatically generated column such as a ROWID or identity column
- Any default values for columns
- All values for an inserted row, without specifying individual column names
- All values that are inserted by a multiple-row INSERT operation
- Values that are changed by a BEFORE INSERT trigger

Example: In addition to examples that use the DB2 sample tables, the examples in this topic use an EMPSAMP table that has the following definition:

```

CREATE TABLE EMPSAMP
(EMPNO     INTEGER GENERATED ALWAYS AS IDENTITY,
NAME       CHAR(30),
SALARY     DECIMAL(10,2),

```

```

DEPTNO    SMALLINT,
LEVEL     CHAR(30),
HIRETYPE  VARCHAR(30) NOT NULL WITH DEFAULT 'New Hire',
HIREDATE  DATE NOT NULL WITH DEFAULT);

```

Assume that you need to insert a row for a new employee into the EMPSAMP table. To find out the values for the generated EMPNO, HIRETYPE, and HIREDATE columns, use the following SELECT FROM INSERT statement:

```

SELECT EMPNO, HIRETYPE, HIREDATE
FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
VALUES('Mary Smith', 35000.00, 11, 'Associate'));

```

The SELECT statement returns the DB2-generated identity value for the EMPNO column, the default value 'New Hire' for the HIRETYPE column, and the value of the CURRENT DATE special register for the HIREDATE column.

Recommendation: Use the SELECT FROM INSERT statement to insert a row into a parent table and retrieve the value of a primary key that was generated by DB2 (a ROWID or identity column). In another INSERT statement, specify this generated value as a value for a foreign key in a dependent table.

Result table of the INSERT operation:

The rows that are inserted into the target table produce a result table whose columns can be referenced in the SELECT list of the query. The columns of the result table are affected by the columns, constraints, and triggers that are defined for the target table:

- The result table includes DB2-generated values for identity columns, ROWID columns, or row change timestamp columns.
- Before DB2 generates the result table, it enforces any constraints that affect the insert operation (that is, check constraints, unique index constraints, and referential integrity constraints).
- The result table includes any changes that result from a BEFORE trigger that is activated by the insert operation. An AFTER trigger does not affect the values in the result table.

Example: Suppose that a BEFORE INSERT trigger is created on table EMPSAMP to give all new employees at the Associate level a \$5000 increase in salary. The trigger has the following definition:

```

CREATE TRIGGER NEW_ASSOC
NO CASCADE BEFORE INSERT ON EMPSAMP
REFERENCING NEW AS NEWSALARY
FOR EACH ROW MODE DB2SQL
WHEN (NEWSALARY.LEVEL = 'ASSOCIATE')
BEGIN ATOMIC
SET NEWSALARY.SALARY = NEWSALARY.SALARY + 5000.00;
END;

```

The INSERT statement in the FROM clause of the following SELECT statement inserts a new employee into the EMPSAMP table:

```

SELECT NAME, SALARY
FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, LEVEL)
VALUES('Mary Smith', 35000.00, 'Associate'));

```

The SELECT statement returns a salary of 40000.00 for Mary Smith instead of the initial salary of 35000.00 that was explicitly specified in the INSERT statement.

Selecting values when you insert a single row:

When you insert a new row into a table, you can retrieve any column in the result table of the SELECT FROM INSERT statement. When you embed this statement in an application, you retrieve the row into host variables by using the SELECT ... INTO form of the statement.

Example: You can retrieve all the values for a row that is inserted into a structure:

```
EXEC SQL SELECT * INTO :empstruct
  FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
    VALUES('Mary Smith', 35000.00, 11, 'Associate'));
```

For this example, :empstruct is a host variable structure that is declared with variables for each of the columns in the EMPSAMP table.

Selecting values when you insert data into a view:

If the INSERT statement references a view that is defined with a search condition, that view must be defined with the WITH CASCADED CHECK OPTION option. When you insert data into the view, the result table of the SELECT FROM INSERT statement includes only rows that satisfy the view definition.

Example: Because view V1 is defined with the WITH CASCADED CHECK OPTION option, you can reference V1 in the INSERT statement:

```
CREATE VIEW V1 AS
  SELECT C1, I1 FROM T1 WHERE I1 > 10
  WITH CASCADED CHECK OPTION;

SELECT C1 FROM
  FINAL TABLE (INSERT INTO V1 (I1) VALUES(12));
```

The value 12 satisfies the search condition of the view definition, and the result table consists of the value for C1 in the inserted row.

If you use a value that does not satisfy the search condition of the view definition, the insert operation fails, and DB2 returns an error.

Selecting values when you insert multiple rows:

In an application program, to retrieve values from the insertion of multiple rows, declare a cursor so that the INSERT statement is in the FROM clause of the SELECT statement of the cursor.

Example: Inserting rows with ROWID values: To see the values of the ROWID columns that are inserted into the employee photo and resume table, you can declare the following cursor:

```
EXEC SQL DECLARE CS1 CURSOR FOR
  SELECT EMP_ROWID
  FROM FINAL TABLE (INSERT INTO DSN8A10.EMP_PHOTO_RESUME (EMPNO)
    SELECT EMPNO FROM DSN8A10.EMP);
```

Example: Using the FETCH FIRST clause: To see only the first five rows that are inserted into the employee photo and resume table, use the FETCH FIRST clause:

```
EXEC SQL DECLARE CS2 CURSOR FOR
  SELECT EMP_ROWID
  FROM FINAL TABLE (INSERT INTO DSN8A10.EMP_PHOTO_RESUME (EMPNO)
    SELECT EMPNO FROM DSN8A10.EMP)
  FETCH FIRST 5 ROWS ONLY;
```

Example: Using the INPUT SEQUENCE clause: To retrieve rows in the order in which they are inserted, use the INPUT SEQUENCE clause:

```
EXEC SQL DECLARE CS3 CURSOR FOR
  SELECT EMP_ROWID
  FROM FINAL_TABLE (INSERT INTO DSN8A10.EMP_PHOTO_RESUME (EMPNO)
                    VALUES (:hva_empno)
                    FOR 5 ROWS)
  ORDER BY INPUT SEQUENCE;
```

The INPUT SEQUENCE clause can be specified only if an INSERT statement is in the FROM clause of the SELECT statement. In this example, the rows are inserted from an array of employee numbers.

Example: Inserting rows with multiple encoding CCSIDs: Suppose that you want to populate an ASCII table with values from an EBCDIC table and then see selected values from the ASCII table. You can use the following cursor to select the EBCDIC columns, populate the ASCII table, and then retrieve the ASCII values:

```
EXEC SQL DECLARE CS4 CURSOR FOR
  SELECT C1, C2
  FROM FINAL_TABLE (INSERT INTO ASCII_TABLE
                   SELECT * FROM EBCDIC_TABLE);
```

Selecting an additional column when you insert data:

You can use the INCLUDE clause to introduce a new column to the result table but not add a column to the target table.

Example: Suppose that you need to insert department number data into the project table. Suppose also, that you want to retrieve the department number and the corresponding manager number for each department. Because MGRNO is not a column in the project table, you can use the INCLUDE clause to include the manager number in your result but not in the insert operation. The following SELECT FROM INSERT statement performs the insert operation and retrieves the data.

```
DECLARE CS1 CURSOR FOR
  SELECT manager_num, projname FROM FINAL_TABLE
  (INSERT INTO PROJ (DEPTNO) INCLUDE(manager_num CHAR(6))
   SELECT DEPTNO, MGRNO FROM DEPT);
```

Result table of the cursor when you insert multiple rows:

In an application program, when you insert multiple rows into a table, you declare a cursor so that the INSERT statement is in the FROM clause of the SELECT statement of the cursor. The result table of the cursor is determined during OPEN cursor processing. The result table may or may not be affected by other processes in your application.

Effect on cursor sensitivity:

When you declare a scrollable cursor, the cursor must be declared with the INSENSITIVE keyword if an INSERT statement is in the FROM clause of the cursor specification. The result table is generated during OPEN cursor processing and does not reflect any future changes. You cannot declare the cursor with the SENSITIVE DYNAMIC or SENSITIVE STATIC keywords.

Effect of searched updates and deletes:

When you declare a non-scrollable cursor, any searched updates or deletes do not affect the result table of the cursor. The rows of the result table are determined during OPEN cursor processing.

Example: Assume that your application declares a cursor, opens the cursor, performs a fetch, updates the table, and then fetches additional rows:

```
EXEC SQL DECLARE CS1 CURSOR FOR
  SELECT SALARY
  FROM FINAL TABLE (INSERT INTO EMP_SAMP (NAME, SALARY, LEVEL)
                    SELECT NAME, INCOME, BAND FROM OLD_EMPLOYEE);
EXEC SQL OPEN CS1;
EXEC SQL FETCH CS1 INTO :hv_salary;
/* print fetch result */
...
EXEC SQL UPDATE EMP_SAMP SET SALARY = SALARY + 500;
while (SQLCODE == 0) {
  EXEC SQL FETCH CS1 INTO :hv_salary;
  /* print fetch result */
  ...
}
```

The fetches that occur after the updates return the rows that were generated when the cursor was opened. If you use a simple SELECT (with no INSERT statement in the FROM clause), the fetches might return the updated values, depending on the access path that DB2 uses.

Effect of WITH HOLD:

When you declare a cursor with the WITH HOLD option and open the cursor, all of the rows are inserted into the target table. The WITH HOLD option has no effect on the SELECT FROM INSERT statement of the cursor definition. After your application performs a commit, you can continue to retrieve all of the inserted rows.

Example: Assume that the employee table in the DB2 sample application has five rows. Your application declares a WITH HOLD cursor, opens the cursor, fetches two rows, performs a commit, and then fetches the third row successfully:

```
EXEC SQL DECLARE CS2 CURSOR WITH HOLD FOR
  SELECT EMP_ROWID
  FROM FINAL TABLE (INSERT INTO DSN8A10.EMP_PHOTO_RESUME (EMPNO)
                    SELECT EMPNO FROM DSN8A10.EMP);
EXEC SQL OPEN CS2; /* Inserts 5 rows */
EXEC SQL FETCH CS2 INTO :hv_rowid; /* Retrieves ROWID for 1st row */
EXEC SQL FETCH CS2 INTO :hv_rowid; /* Retrieves ROWID for 2nd row */
EXEC SQL COMMIT; /* Commits 5 rows */
EXEC SQL FETCH CS2 INTO :hv_rowid; /* Retrieves ROWID for 3rd row */
```

Effect of SAVEPOINT and ROLLBACK:

A savepoint is a point in time within a unit of recovery to which relational database changes can be rolled back. You can set a savepoint with the SAVEPOINT statement.

When you set a savepoint prior to opening the cursor and then roll back to that savepoint, all of the insertions are undone.

Example: Assume that your application declares a cursor, sets a savepoint, opens the cursor, sets another savepoint, rolls back to the second savepoint, and then rolls back to the first savepoint:

```

EXEC SQL DECLARE CS3 CURSOR FOR
    SELECT EMP_ROWID
    FROM FINAL TABLE (INSERT INTO DSN8A10.EMP_PHOTO_RESUME (EMPNO)
        SELECT EMPNO FROM DSN8A10.EMP);
EXEC SQL SAVEPOINT A ON ROLLBACK RETAIN CURSORS;      /* Sets 1st savepoint */
EXEC SQL OPEN CS3;
EXEC SQL SAVEPOINT B ON ROLLBACK RETAIN CURSORS;      /* Sets 2nd savepoint */
...
EXEC SQL ROLLBACK TO SAVEPOINT B; /* Rows still in DSN8A10.EMP_PHOTO_RESUME */
...
EXEC SQL ROLLBACK TO SAVEPOINT A; /* All inserted rows are undone */

```

What happens if an error occurs: In an application program, when you insert one or more rows into a table by using the `SELECT FROM INSERT` statement, the result table of the insert operation may or may not be affected, depending on where the error occurred in the application processing.

During SELECT INTO processing: If the insert processing or the select processing fails during a `SELECT INTO` statement, no rows are inserted into the target table, and no rows are returned from the result table of the insert operation.

Example: Assume that the employee table of the DB2 sample application has one row, and that the `SALARY` column has a value of 9 999 000.00.

```

EXEC SQL SELECT EMPNO INTO :hv_empno
    FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY)
        SELECT FIRSTNAME || MIDINIT || LASTNAME,
            SALARY + 10000.00
        FROM DSN8A10.EMP)

```

The addition of 10000.00 causes a decimal overflow to occur, and no rows are inserted into the `EMPSAMP` table.

During OPEN cursor processing: If the insertion of any row fails during the `OPEN` cursor processing, all previously successful insertions are undone. The result table of the insert is empty.

During FETCH processing: If the `FETCH` statement fails while retrieving rows from the result table of the insert operation, a negative `SQLCODE` is returned to the application, but the result table still contains the original number of rows that was determined during the `OPEN` cursor processing. At this point, you can undo all of the inserts.

Example: Assume that the result table contains 100 rows and the 90th row that is being fetched from the cursor returns a negative `SQLCODE`:

```

EXEC SQL DECLARE CS1 CURSOR FOR
    SELECT EMPNO
    FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY)
        SELECT FIRSTNAME || MIDINIT || LASTNAME, SALARY + 10000.00
        FROM DSN8A10.EMP);
EXEC SQL OPEN CS1; /* Inserts 100 rows */
while (SQLCODE == 0)
    EXEC SQL FETCH CS1 INTO :hv_empno;
if (SQLCODE == -904) /* If SQLCODE is -904, undo all inserts */
    EXEC SQL ROLLBACK;
else /* Else, commit inserts */
    EXEC SQL COMMIT;

```

Related concepts:

“Held and non-held cursors” on page 725

“Rules for host variables in an SQL statement” on page 183

“Identity columns” on page 466

“Types of cursors” on page 723

Related tasks:

“Inserting multiple rows of data from host variable arrays” on page 192

“Retrieving a set of rows by using a cursor” on page 722

“Undoing selected changes within a unit of work by using savepoints” on page 65

Related reference:

“Command line processor BIND command” on page 946

Preserving the order of a derived table

When you specify `SELECT FROM INSERT`, `SELECT FROM UPDATE`, `SELECT FROM DELETE`, or `SELECT FROM MERGE`, you can preserve the order of the derived table. This action ensures that the result rows of a fullselect follow the same order as the result table of a subquery within the fullselect.

To preserve the order of the derived table specify the `ORDER OF` clause with the `ORDER BY` clause. These two clauses ensure that the result rows of a fullselect follow the same order as the result table of a subquery within the fullselect.

You can use the `ORDER OF` clause in any query that uses an `ORDER BY` clause, but the `ORDER OF` clause is most useful with queries that contain a set operator, such as `UNION`.

Example: The following example retrieves the following rows:

- Rows of table T1 in no specified order
- Rows of table T2 in the order of the first column in table T2

The example query then performs a `UNION ALL` operation on the results of the two subqueries. The `ORDER BY ORDER OF UTABLE` clause in the query specifies that the fullselect result rows are to be returned in the same order as the result rows of the `UNION ALL` statement.

```
SELECT * FROM
  (SELECT * FROM T1
   UNION ALL
   (SELECT * FROM T2 ORDER BY 1)
  ) AS UTABLE
ORDER BY ORDER OF UTABLE;
```

Example: The following example joins data from table T1 to the result table of a nested table expression. The nested table expression is ordered by the second column in table T2. The `ORDER BY ORDER OF TEMP` clause in the query specifies that the fullselect result rows are to be returned in the same order as the nested table expression.

```
SELECT T1.C1, T1.C2, TEMP.Cy, TEMP.Cx
FROM T1, (SELECT T2.C1, T2.C2 FROM T2 ORDER BY 2) as TEMP(Cx, Cy)
WHERE Cy = T1.C1
ORDER BY ORDER OF TEMP;
```

Alternatively, you can produce the same result by explicitly stating the `ORDER BY` column `TEMP.Cy` in the fullselect instead of using the `ORDER OF` syntax.

```
SELECT T1.C1, T1.C2, TEMP.Cy, TEMP.Cx
FROM T1, (SELECT T2.C1, T2.C2 FROM T2 ORDER BY 2) as TEMP(Cx, Cy)
WHERE Cy = T1.C1
ORDER BY TEMP.Cy;
```

Adding data to the end of a table

In a relational database, the rows of a table are not ordered, and thus, the table has no “end.” However, depending on your goal, you can perform several actions to simulate adding data to the end of a table.

Question: How can I add data to the end of a table?

Answer: Though the question is often asked, it has no meaning in a relational database. The rows of a base table are not ordered; hence, the table does not have an “end”.

However, depending on your goal, you can perform one of the following actions to simulate adding data to the end of a table:

- If your goal is to get a result table that is ordered according to when the rows were inserted, define a unique index on a `TIMESTAMP` column in the table definition. Then, when you retrieve data from the table, use an `ORDER BY` clause that names that column. The newest insert appears last.
- If your goal is for DB2 to insert rows in the next available free space, without preserving clustering order, specify the `APPEND YES` option when you create or alter the table. Specifying this option might reduce the time it takes to insert rows, because DB2 does not spend time searching for free space.

Storing data that does not have a tabular format

DB2 provides several options for you to store large volumes of data that is not defined as a set of columns in a table.

Question: How can I store a large volume of data that is not defined as a set of columns in a table?

Answer: You can store the data in a table in a binary string, a LOB, or an XML column.

Updating table data

You can change a column value to another value or remove the column value altogether.

To change the data in a table, use the `UPDATE` statement. You can also use the `UPDATE` statement to remove a value from a column (without removing the row) by changing the column value to null.

Example: Suppose that an employee relocates. To update several items of the employee's data in the `YEMP` work table to reflect the move, you can execute the following statement:

```
UPDATE YEMP
  SET JOB = 'MANAGER ',
      PHONENO = '5678'
  WHERE EMPNO = '000400';
```

You cannot update rows in a created temporary table, but you can update rows in a declared temporary table.

The SET clause names the columns that you want to update and provides the values that you want to assign to those columns. You can replace a column value in the SET clause with any of the following items:

- A null value
The column to which you assign the null value must not be defined as NOT NULL.
- An expression, which can be any of the following items:
 - A column
 - A constant
 - A scalar fullselect
 - A host variable
 - A special register
- A default value
If you specify DEFAULT, DB2 determines the value based on how the corresponding column is defined in the table.

In addition, you can replace one or more column values in the SET clause with the column values in a row that is returned by a fullselect.

Next, identify the rows to update:

- To update a single row, use a WHERE clause that locates one, and only one, row.
- To update several rows, use a WHERE clause that locates only the rows that you want to update.

If you omit the WHERE clause, DB2 updates **every row** in the table or view with the values that you supply.

If DB2 finds an error while executing your UPDATE statement (for example, an update value that is too large for the column), it stops updating and returns an error. No rows in the table change. Rows that were already changed, if any, are restored to their previous values. If the UPDATE statement is successful, SQLERRD(3) is set to the number of rows that are updated.

Example: The following statement supplies a missing middle initial and changes the job for employee 000200.

```
UPDATE YEMP
  SET MIDINIT = 'H', JOB = 'FIELDREP'
  WHERE EMPNO = '000200';
```

The following statement gives everyone in department D11 a raise of 400.00. The statement can update several rows.

```
UPDATE YEMP
  SET SALARY = SALARY + 400.00
  WHERE WORKDEPT = 'D11';
```

The following statement sets the salary for employee 000190 to the average salary and sets the bonus to the minimum bonus for all employees.

```
UPDATE YEMP
  SET (SALARY, BONUS) =
  (SELECT AVG(SALARY), MIN(BONUS)
   FROM EMP)
  WHERE EMPNO = '000190';
```

Selecting values while updating data

When you update rows in a table, you can select the updated values from those rows at the same time.

You can select values from rows that are being updated by specifying the UPDATE statement in the FROM clause of the SELECT statement. When you update one or more rows in a table, you can retrieve:

- The value of an automatically generated column such as a ROWID or identity column
- Any default values for columns
- All values for an updated row, without specifying individual column names

In most cases, you want to use the FINAL TABLE clause with SELECT FROM UPDATE statements. The FINAL TABLE consists of the rows of the table or view after the update occurs.

Example: Suppose that all clerks for a company are receiving 5 percent raises. You can use the following SELECT FROM UPDATE statement to increase the salary of each designer by 5 percent and to retrieve the total increase in salary for the company.

```
SELECT SUM(SALARY) INTO :salary FROM FINAL TABLE
  (UPDATE EMP SET SALARY = SALARY * 1.05
   WHERE JOB = 'DESIGNER');
```

To retrieve row-by-row output of updated data, use a cursor with a SELECT FROM UPDATE statement.

Example: Suppose that all designers for a company are receiving a 30 percent increase in their bonus. You can use the following SELECT FROM UPDATE statement to increase the bonus of each clerk by 30 percent and to retrieve the bonus for each clerk.

```
DECLARE CS1 CURSOR FOR
  SELECT LASTNAME, BONUS FROM FINAL TABLE
  (UPDATE EMP SET BONUS = BONUS * 1.3
   WHERE JOB = 'CLERK');
FETCH CS1 INTO :lastname, :bonus;
```

You can use the INCLUDE clause to introduce a new column to the result table but not add the column to the target table.

Example: Suppose that sales representatives received a 20 percent increase in their commission. You need to update the commission (COMM) of sales representatives (SALESREP) in the EMP table and that you need to retrieve the old commission and the new commission for each sales representative. You can use the following SELECT FROM UPDATE statement to perform the update and to retrieve the required data.

```
DECLARE CS2 CURSOR FOR
  SELECT LASTNAME, COMM, old_comm FROM FINAL TABLE
  (UPDATE EMP INCLUDE(old_comm DECIMAL (7,2))
   SET COMM = COMM * 1.2, old_comm = COMM
   WHERE JOB = 'SALESREP');
```

Updating thousands of rows

When you update large volumes of data, consider certain recommended actions to increase concurrency.

Question: Are there any special techniques for updating large volumes of data?

Answer: Yes. When updating large volumes of data using a cursor, you can minimize the amount of time that you hold locks on the data by declaring the cursor with the HOLD option and by issuing commits frequently.

Deleting data from tables

You can delete data from a table by deleting one or more rows from the table or by deleting all rows from the table.

To delete one or more rows in a table:

- Use the DELETE statement with a WHERE clause to specify a search condition.

The DELETE statement removes zero or more rows of a table, depending on how many rows satisfy the search condition that you specify in the WHERE clause.

You can use DELETE with a WHERE clause to remove only selected rows from a declared temporary table, but not from a created temporary table.

The following DELETE statement deletes each row in the YEMP table that has an employee number '000060'.

```
DELETE FROM YEMP
WHERE EMPNO = '000060';
```

When this statement executes, DB2 deletes any row from the YEMP table that meets the search condition.

If DB2 finds an error while executing your DELETE statement, it stops deleting data and returns error codes in the SQLCODE and SQLSTATE variables or related fields in the SQLCA. The data in the table does not change.

If the DELETE is successful, SQLERRD(3) in the SQLCA contains the number of deleted rows. This number includes only the number of deleted rows in the table that is specified in the DELETE statement. Rows that are deleted (in other tables) according to the CASCADE rule are not included in SQLERRD(3).

To delete every row in a table:

- Use the DELETE statement without specifying a WHERE clause.

With segmented table spaces, deleting all rows of a table is very fast.

The following DELETE statement deletes every row in the YDEPT table:

```
DELETE FROM YDEPT;
```

If the statement executes, the table continues to exist (that is, you can insert rows into it), but it is empty. All existing views and authorizations on the table remain intact when using DELETE.

- Use the TRUNCATE statement.

The TRUNCATE statement can provide the following advantages over a DELETE statement:

- The TRUNCATE statement can ignore delete triggers
- The TRUNCATE statement can perform an immediate commit
- The TRUNCATE statement can keep storage allocated for the table

The TRUNCATE statement does not, however, reset the count for an automatically generated value for an identity column on the table. If 14872 was the next identity column value to be generated before a TRUNCATE statement, 14872 would be the next value generated after the TRUNCATE statement.

Suppose that you need to empty the data from an old inventory table, regardless of any existing delete triggers, and you need to make the space that is allocated for the table available for other uses. Use the following TRUNCATE statement.

```
TRUNCATE INVENTORY_TABLE
  IGNORE DELETE TRIGGERS
  DROP STORAGE;
```

Suppose that you need to empty the data from an old inventory table permanently, regardless of any existing delete triggers, and you need to preserve the space that is allocated for the table. You need the emptied data to be completely unavailable, so that a ROLLBACK statement cannot return the data. Use the following TRUNCATE statement.

```
TRUNCATE INVENTORY_TABLE
  REUSE STORAGE
  IGNORE DELETE TRIGGERS
  IMMEDIATE;
```

- Use the DROP TABLE statement.

DROP TABLE drops the specified table and all related views and authorizations, which can invalidate plans and packages.

Related concepts:

[➤](#) SQL communication area (SQLCA) (DB2 SQL)

Related tasks:

“Dropping tables” on page 484

Related reference:

[➤](#) DELETE (DB2 SQL)

[➤](#) DROP (DB2 SQL)

[➤](#) TRUNCATE (DB2 SQL)

Selecting values while deleting data

When you delete rows from a table, you can select the values from those rows at the same time.

You can select values from rows that are being deleted by specifying the DELETE statement in the FROM clause of the SELECT statement. When you delete one or more rows in a table, you can retrieve:

- Any default values for columns
- All values for a deleted row, without specifying individual column names
- Calculated values based on deleted rows

When you use a SELECT FROM DELETE statement, you must use the FROM OLD TABLE clause to retrieve deleted values. The OLD TABLE consists of the rows of the table or view before the delete occurs.

Example: Suppose that a company is eliminating all operator positions and that the company wants to know how much salary money it will save by eliminating these positions. You can use the following SELECT FROM DELETE statement to delete operators from the EMP table and to retrieve the sum of operator salaries.

```
SELECT SUM(SALARY) INTO :salary FROM OLD TABLE
  (DELETE FROM EMP
   WHERE JOB = 'OPERATOR');
```

To retrieve row-by-row output of deleted data, use a cursor with a SELECT FROM DELETE statement.

Example: Suppose that a company is eliminating all analyst positions and that the company wants to know how many years of experience each analyst had with the

| company. You can use the following SELECT FROM DELETE statement to delete
| analysts from the EMP table and to retrieve the experience of each analyst.

```
| DECLARE CS1 CURSOR FOR  
| SELECT YEAR(CURRENT DATE - HIREDATE) FROM OLD TABLE  
| (DELETE FROM EMP  
| WHERE JOB = 'ANALYST');  
| FETCH CS1 INTO :years_of_service;
```

| If you need to retrieve calculated data based on the data that you delete but not
| add that column to the target table.

| **Example:** Suppose that you need to delete managers from the EMP table and that
| you need to retrieve the salary and the years of employment for each manager.
| You can use the following SELECT FROM DELETE statement to perform the delete
| operation and to retrieve the required data.

```
| DECLARE CS2 CURSOR FOR  
| SELECT LASTNAME, SALARY, years_employed FROM OLD TABLE  
| (DELETE FROM EMP INCLUDE(years_employed INTEGER)  
| SET years_employed = YEAR(CURRENT DATE - HIREDATE)  
| WHERE JOB = 'MANAGER');
```

Chapter 12. Accessing data

Your program can use a number of different techniques to read data from any DB2 tables for which you have read access. The simplest technique is to use basic SQL SELECT statements. However, you should choose the technique that works best for your situation and performs well.

Related concepts:

- Investigating SQL performance by using EXPLAIN (DB2 Performance)
- Interpreting data access by using EXPLAIN (DB2 Performance)

Related tasks:

- Writing efficient SQL queries (DB2 Performance)
- Generating visual representations of access plans (IBM Data Studio)

Determining which tables you have access to

You can ask DB2 to list the tables that a specific authorization ID has access to.

The contents of the DB2 catalog tables can be a useful reference tool when you begin to develop an SQL statement or an application program.

The catalog table, SYSIBM.SYSTABAUTH, lists table privileges that are granted to authorization IDs. To display the tables that you have authority to access (by privileges granted either to your authorization ID or to PUBLIC), you can execute an SQL statement similar to the one shown in the following example. To do this, you must have the SELECT privilege on SYSIBM.SYSTABAUTH.

Example: The following statement displays the tables that the current user has authority to access:

```
SELECT DISTINCT TCREATOR, TTNAME
  FROM SYSIBM.SYSTABAUTH
 WHERE GRANTEE IN (USER, 'PUBLIC', 'PUBLIC*') AND GRANTEETYPE = ' ';
```

In this query, the predicate GRANTEETYPE = ' ' selects authorization IDs.

Exception: If your DB2 subsystem uses an exit routine for access control authorization, you cannot rely on catalog queries to tell you the tables that you can access. When such an exit routine is installed, both RACF and DB2 control table access.

Displaying information about the columns for a given table

You can ask DB2 to list the columns in a particular table and certain information about those columns.

The catalog table, SYSIBM.SYSCOLUMNS, describes every column of every table.

Example: Suppose that you want to display information about table DSN8A10.DEPT. If you have the SELECT privilege on SYSIBM.SYSCOLUMNS, you can use the following statement:

```

SELECT NAME, COLTYPE, SCALE, LENGTH
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBNAME = 'DEPT'
    AND TBCREATOR = 'DSN8A10';

```

If you display column information about a table that includes LOB or ROWID columns, the LENGTH field for those columns contains the number of bytes that those column occupy in the base table. The LENGTH field does not contain the length of the LOB or ROWID data.

Example: To determine the maximum length of data for a LOB or ROWID column, include the LENGTH2 column in your query:

```

SELECT NAME, COLTYPE, LENGTH, LENGTH2
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBNAME = 'EMP_PHOTO_RESUME'
    AND TBCREATOR = 'DSN8A10';

```

Retrieving data by using the SELECT statement

The simplest way to retrieve data is to use the SQL SELECT statement to specify a result table. You can specify the columns and rows that you want to retrieve.

Consider developing SQL statements similar to the examples in this section, and then running them dynamically using SPUFI, the command line processor, or DB2 Query Management Facility (DB2 QMF).

You do not need to know the column names to select DB2 data. Use an asterisk (*) in the SELECT clause to indicate that you want to retrieve all columns of each selected row of the named table. Implicitly hidden columns, such as ROWID columns and XML document ID columns, are not included in the result of the SELECT * statement. To view the values of these columns, you must specify the column name.

Example: SELECT *: The following SQL statement selects all columns from the department table:

```

SELECT *
  FROM DSN8A10.DEPT;

```

The result table looks similar to the following output:

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
=====	=====	=====	=====	=====
A00	SPIFFY COMPUTER SERVICES DIV.	000010	A00	-----
B01	PLANNING	000020	A00	-----
C01	INFORMATION CENTER	000030	A00	-----
D01	DEVELOPMENT CENTER	-----	A00	-----
D11	MANUFACTURING CENTER	000060	D01	-----
D21	ADMINISTRATION SYSTEMS	000070	D01	-----
E01	SUPPORT SERVICES	000050	A00	-----
E11	OPERATIONS	000090	E01	-----
E21	SOFTWARE SUPPORT	000100	E01	-----
F22	BRANCH OFFICE F2	-----	E01	-----
G22	BRANCH OFFICE G2	-----	E01	-----
H22	BRANCH OFFICE H2	-----	E01	-----
I22	BRANCH OFFICE I2	-----	E01	-----
J22	BRANCH OFFICE J2	-----	E01	-----

Because the example does not specify a WHERE clause, the statement retrieves data from all rows.

The dashes for MGRNO and LOCATION in the result table indicate null values.

SELECT * is recommended mostly for use with dynamic SQL and view definitions. You can use SELECT * in static SQL, but doing so is not recommended because of host variable compatibility and performance reasons. Suppose that you add a column to the table to which SELECT * refers. If you have not defined a receiving host variable for that column, an error occurs.

If you list the column names in a static SELECT statement instead of using an asterisk, you can avoid the problem that sometimes occurs with SELECT *. You can also see the relationship between the receiving host variables and the columns in the result table.

Selecting some columns: SELECT column-name:

Select the column or columns you want to retrieve by naming each column. All columns appear in the order you specify, not in their order in the table.

Example: SELECT column-name: The following SQL statement retrieves only the MGRNO and DEPTNO columns from the department table:

```
SELECT MGRNO, DEPTNO
       FROM DSN8A10.DEPT;
```

The result table looks similar to the following output:

MGRNO	DEPTNO
=====	=====
000010	A00
000020	B01
000030	C01
-----	D01
000050	E01
000060	D11
000070	D21
000090	E11
000100	E21
-----	F22
-----	G22
-----	H22
-----	I22
-----	J22

With a single SELECT statement, you can select data from one column or as many as 750 columns.

To SELECT data from implicitly hidden columns, such as ROWID and XML document ID, look up the column names in SYSIBM.SYSCOLUMNS and specify these names in the SELECT list. For example, suppose that you create and populate the following table:

```
CREATE TABLE MEMBERS (MEMBERID INTEGER,
                       BIO XML,
                       REPORT XML,
                       RECOMMENDATIONS XML);
```

DB2 generates one additional implicitly hidden XML document ID column. To retrieve data in all columns, including the generated XML document ID column, first look up the name of the generated column in SYSIBM.SYSCOLUMNS. Suppose the name is DB2_GENERATED_DOCID_FOR_XML. Then, specify the following statement:

```
SELECT DB2_GENERATED_DOCID_FOR_XML, MEMBERID, BIO,
       REPORT, RECOMMENDATIONS FROM MEMBERS
```

Selecting rows using search conditions: WHERE:

Use a WHERE clause to select the rows that meet certain conditions. A WHERE clause specifies a search condition. A *search condition* consists of one or more predicates. A *predicate* specifies a test that you want DB2 to apply to each table row.

DB2 evaluates a predicate for each row as true, false, or unknown. Results are unknown only if an operand is null.

If a search condition contains a column of a distinct type, the value to which that column is compared must be of the same distinct type, or you must cast the value to the distinct type.

The following table lists the type of comparison, the comparison operators, and an example of each type of comparison that you can use in a predicate in a WHERE clause.

Table 109. Comparison operators used in conditions

Type of comparison	Comparison operator	Example
Equal to	=	DEPTNO = 'X01'
Not equal to	<>	DEPTNO <> 'X01'
Less than	<	AVG(SALARY) < 30000
Less than or equal to	<=	AGE <= 25
Not less than	>=	AGE >= 21
Greater than	>	SALARY > 2000
Greater than or equal to	>=	SALARY >= 5000
Not greater than	<=	SALARY <= 5000
Equal to null	IS NULL	PHONENO IS NULL
Not equal to another value or one value is equal to null	IS DISTINCT FROM	PHONENO IS DISTINCT FROM :PHONEHV
Similar to another value	LIKE	NAME LIKE ' or STATUS LIKE 'N_'
At least one of two conditions	OR	HIREDATE < '1965-01-01' OR SALARY < 16000
Both of two conditions	AND	HIREDATE < '1965-01-01' AND SALARY < 16000
Between two values	BETWEEN	SALARY BETWEEN 20000 AND 40000
Equals a value in a set	IN (X, Y, Z)	DEPTNO IN ('B01', 'C01', 'D01')

Note: SALARY BETWEEN 20000 AND 40000 is equivalent to SALARY >= 20000 AND SALARY <= 40000.

You can also search for rows that **do not** satisfy one of the preceding conditions by using the NOT keyword before the specified condition.

You can search for rows that do not satisfy the IS DISTINCT FROM predicate by using either of the following predicates:

- *value 1 IS NOT DISTINCT FROM value 2*
- *NOT(value 1 IS DISTINCT FROM value 2)*

Both of these forms of the predicate create an expression for which one value is equal to another value or both values are equal to null.

Related concepts:

“Distinct types” on page 513

“Host variables” on page 174

“Remote servers and distributed data” on page 68

“Subqueries” on page 710

 [Predicates \(DB2 SQL\)](#)

Related tasks:

 [Coding SQL statements to avoid unnecessary processing \(DB2 Performance\)](#)

Selecting derived columns

In an SQL SELECT statement, you can select columns that are not actual columns in a table. Instead, you can specify “columns” that are derived from a constant, an expression, or a function.

Example: SELECT with an expression: This SQL statement generates a result table in which the second column is a derived column that is generated by adding the values of the SALARY, BONUS, and COMM columns.

```
SELECT EMPNO, (SALARY + BONUS + COMM)
FROM DSN8A10.EMP;
```

Derived columns in a result table, such as (SALARY + BONUS + COMM), do not have names. You can use the AS clause to give a name to an unnamed column of the result table. For information about using the AS clause, see “Naming result columns” on page 683.

To order the rows in a result table by the values in a derived column, specify a name for the column by using the AS clause, and specify that name in the ORDER BY clause. For information about using the ORDER BY clause, see “Ordering the result table rows” on page 684.

Selecting XML data

You can select all XML data that is stored in a particular column or only a subset of data from an XML column.

You can select all XML data that is stored in a particular column by specifying SELECT *column name* or SELECT *, just as you would for columns of any other data type. Alternatively, you can select only a subset of data from an XML column by using an XPath expression in a SELECT statement. XPath expressions identify specific nodes in an XML document.

To select a subset of data in an XML column, specify the XMLQUERY function in your SELECT statement with the following parameters:

- An XPath expression that is embedded in a character string constant. Specify an XPath expression that identifies which XML data to return.
- Any additional values to pass to the XPath expression, including the XML column name. Specify these values after the PASSING keyword.

Example: Suppose that you store purchase orders as XML documents in the POrder column in the PurchaseOrders table. You need to find in each purchase order the items whose product name is equal to a name in the Product table. You can use the following statement to find these values:

```
SELECT XMLQUERY('//*[productName = $n]'
  PASSING PO.POrder,
  P.name AS "n")
FROM PurchaseOrders PO, Product P;
```

This statement returns the item elements in the POrder column that satisfy the criteria in the XPath expression.

Related concepts:

[➡ Overview of XQuery \(DB2 Programming for XML\)](#)

Related reference:

[➡ XMLQUERY \(DB2 SQL\)](#)

Formatting the result table

An SQL statement returns data in a table called a result table. You can specify certain attributes of the result table, such as the column names, how the rows are ordered, and whether the rows are numbered.

Result tables

The data that is retrieved by an SQL statement is always in the form of a table, which is called a *result table*. Like the tables from which you retrieve the data, a result table has rows and columns. A program fetches this data one row at a time.

Example result table: Assume that you issue the following SELECT statement, which retrieves the last name, first name, and phone number of employees in department D11 from the sample employee table:

```
SELECT LASTNAME, FIRSTNME, PHONENO
  FROM DSN8A10.EMP
  WHERE WORKDEPT = 'D11'
  ORDER BY LASTNAME;
```

The result table looks similar to the following output:

LASTNAME	FIRSTNME	PHONENO
ADAMSON	BRUCE	4510
BROWN	DAVID	4501
JOHN	REBA	0672
JONES	WILLIAM	0942
LUTZ	JENNIFER	0672
PIANKA	ELIZABETH	3782
SCOUTTEN	MARILYN	1682
STERN	IRVING	6432
WALKER	JAMES	2986
YAMAMOTO	KIYOSHI	2890
YOSHIMURA	MASATOSHI	2890

Eliminating redundant duplicate rows in the result table

If a query result table contains multiple identical rows, you can ask DB2 to remove these redundant rows. For example, a query might return multiple rows for each employee when one row per employee is sufficient for your program.

The DISTINCT keyword removes redundant duplicate rows from your result table, so that each row contains unique data.

Example: SELECT DISTINCT: The following SELECT statement lists unique department numbers for administrative departments:

```
SELECT DISTINCT ADMRDEPT
  FROM DSN8A10.DEPT;
```

The result table looks similar to the following output:

```
ADMRDEPT
=====
A00
D01
E01
```

Restriction: You cannot use the DISTINCT keyword with LOB columns or XML columns.

Related tasks:

 [Coding SQL statements to avoid unnecessary processing \(DB2 Performance\)](#)

Related reference:

 [select-clause \(DB2 SQL\)](#)

Naming result columns

You can provide your own names for the result table columns for a SELECT statement. This capability is particularly useful for a column that is derived from an expression or a function.

With the AS clause, you can name result columns in a SELECT statement.

The following examples show different ways to use the AS clause.

Example: SELECT with AS CLAUSE: The following example of the SELECT statement gives the expression SALARY+BONUS+COMM the name TOTAL_SAL.

```
SELECT SALARY+BONUS+COMM AS TOTAL_SAL
  FROM DSN8A10.EMP
  ORDER BY TOTAL_SAL;
```

Example: CREATE VIEW with AS clause: You can specify result column names in the select-clause of a CREATE VIEW statement. You do not need to supply the column list of CREATE VIEW, because the AS keyword names the derived column. The columns in the view EMP_SAL are EMPNO and TOTAL_SAL.

```
CREATE VIEW EMP_SAL AS
  SELECT EMPNO,SALARY+BONUS+COMM AS TOTAL_SAL
  FROM DSN8A10.EMP;
```

Example: set operator with AS clause: You can use the AS clause with set operators, such as UNION. In this example, the AS clause is used to give the same name to corresponding columns of tables in a UNION. The third result column from the union of the two tables has the name TOTAL_VALUE, even though it contains data that is derived from columns with different names:

```
SELECT 'On hand' AS STATUS, PARTNO, QOH * COST AS TOTAL_VALUE
  FROM PART_ON_HAND
UNION ALL
SELECT 'Ordered' AS STATUS, PARTNO, QORDER * COST AS TOTAL_VALUE
  FROM ORDER_PART
ORDER BY PARTNO, TOTAL_VALUE;
```

The column STATUS and the derived column TOTAL_VALUE have the same name in the first and second result tables. They are combined in the union of the two result tables, which is similar to the following partial output:

STATUS	PARTNO	TOTAL_VALUE
=====	=====	=====
On hand	00557	345.60
Ordered	00557	150.50
.		
.		
.		

Example: GROUP BY derived column: You can use the AS clause in a FROM clause to assign a name to a derived column that you want to refer to in a GROUP BY clause. This SQL statement names HIREYEAR in the nested table expression, which lets you use the name of that result column in the GROUP BY clause:

```
SELECT HIREYEAR, AVG(SALARY)
      FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
            FROM DSN8A10.EMP) AS NEWEMP
      GROUP BY HIREYEAR;
```

You cannot use GROUP BY with a name that is defined with an AS clause for the derived column YEAR(HIREDATE) in the outer SELECT, because that name does not exist when the GROUP BY runs. However, you can use GROUP BY with a name that is defined with an AS clause in the nested table expression, because the nested table expression runs before the GROUP BY that references the name.

Related tasks:

- “Combining result tables from multiple SELECT statements” on page 688
- “Defining a view” on page 485
- “Summarizing group values” on page 692

Related reference:

 [select-clause \(DB2 SQL\)](#)

Ordering the result table rows

If you want to guarantee that the rows in your result table are ordered in a particular way, you must specify the order in the SELECT statement. Otherwise, DB2 can return the rows in any order.

To retrieve rows in a specific order, use the ORDER BY clause. Using ORDER BY is the only way to guarantee that your rows are ordered as you want them. The following topics show you how to use the ORDER BY clause.

Specifying the sort key in the ORDER BY clause:

The order of the selected rows depends on the sort keys that you identify in the ORDER BY clause. A *sort key* can be a column name, an integer that represents the number of a column in the result table, or an expression. DB2 orders the rows by the first sort key, followed by the second sort key, and so on.

You can list the rows in ascending or descending order. Null values appear last in an ascending sort and first in a descending sort.

DB2 sorts strings in the collating sequence associated with the encoding scheme of the table. DB2 sorts numbers algebraically and sorts datetime values chronologically.

Restriction: You cannot use the ORDER BY clause with LOB or XML columns.

Example: ORDER BY clause with a column name as the sort key: Retrieve the employee numbers, last names, and hire dates of employees in department A00 in ascending order of hire dates:

```
SELECT EMPNO, LASTNAME, HIREDATE
       FROM DSN8A10.EMP
       WHERE WORKDEPT = 'A00'
       ORDER BY HIREDATE ASC;
```

The result table looks similar to the following output:

EMPNO	LASTNAME	HIREDATE
000110	LUCCHESI	1958-05-16
000120	O'CONNELL	1963-12-05
000010	HAAS	1965-01-01
200010	HEMMINGER	1965-01-01
200120	ORLANDO	1972-05-05

Example: ORDER BY clause with an expression as the sort key: The following subselect retrieves the employee numbers, salaries, commissions, and total compensation (salary plus commission) for employees with a total compensation greater than 40000. Order the results by total compensation:

```
SELECT EMPNO, SALARY, COMM, SALARY+COMM AS "TOTAL COMP"
       FROM DSN8A10.EMP
       WHERE SALARY+COMM > 40000
       ORDER BY SALARY+COMM;
```

The intermediate result table looks similar to the following output:

EMPNO	SALARY	COMM	TOTAL COMP
000030	38250.00	3060.00	41310.00
000050	40175.00	3214.00	43389.00
000020	41250.00	3300.00	44550.00
000110	46500.00	3720.00	50220.00
200010	46500.00	4220.00	50720.00
000010	52750.00	4220.00	56970.00

Referencing derived columns in the ORDER BY clause:

If you use the AS clause to name an unnamed column in a SELECT statement, you can use that name in the ORDER BY clause.

Example: ORDER BY clause that uses a derived column: The following SQL statement orders the selected information by total salary:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
       FROM DSN8A10.EMP
       ORDER BY TOTAL_SAL;
```

Numbering the rows in a result table

DB2 does not number the rows in the result table for a query unless you explicitly request that the rows be numbered.

To number the rows in a result table, include the ROW_NUMBER specification in your query. If you want to ensure that the rows are in a particular order, include an ORDER BY clause after the OVER keyword. Otherwise, the rows are numbered in an arbitrary order.

Suppose that you want a list of employees and salaries from department D11 in the sample EMP table. You can return a numbered list that is ordered by last name by submitting the following query:

```
SELECT ROW_NUMBER() OVER (ORDER BY LASTNAME) AS NUMBER,
WORKDEPT, LASTNAME, SALARY
FROM DSN8910.EMP
WHERE WORKDEPT='D11'
```

This query returns the following result:

NUMBER	WORKDEPT	LASTNAME	SALARY
1	D11	ADAMSON	25280.00
2	D11	BROWN	27740.00
3	D11	JOHN	29840.00
4	D11	JONES	18270.00
5	D11	LUTZ	29840.00
6	D11	PIANKA	22250.00
7	D11	SCOUTTEN	21340.00
8	D11	STERN	32250.00
9	D11	WALKER	20450.00
10	D11	YAMAMOTO	24680.00
11	D11	YOSHIMURA	24680.00

Related reference:

 OLAP specification (DB2 SQL)

Ranking the rows

You can request that DB2 calculate the ordinal rank of each row in the result set based on a particular column. For example, you can rank finishing times for a marathon to determine the first, second, and third place finishers.

To rank rows, use one of the following ranking specifications in an SQL statement:

RANK

Returns a rank number for each row value. Use this specification if you want rank numbers to be skipped when duplicate row values exist. For example, suppose the top five finishers in a marathon have the following times:

- 2:31:57
- 2:34:52
- 2:34:52
- 2:37:26
- 2:38:01

When you use the RANK specification, DB2 returns the following rank numbers:

Table 110. Example of values returned when you specify RANK

Value	Rank number
2:31:57	1
2:34:52	2
2:34:52	2
2:37:26	4
2:38:01	5

DENSE_RANK

Returns a rank number for each row value. Use this specification if you do not want rank numbers to be skipped when duplicate row values exist. For example, when you specify DENSE_RANK with the same times that are listed in the description of RANK, DB2 returns the following rank numbers:

Table 111. Example of values returned when you specify RANK

Value	Rank number
2:31:57	1
2:34:52	2
2:34:52	2
2:37:26	3
2:38:01	4

Example: Suppose that you had the following values in the DATA column of table T1:

```
DATA
-----
100
 35
 23
  8
  8
  6
```

Suppose that you use the following RANK specification:

```
SELECT DATA,
       RANK() OVER (ORDER BY DATA DESC) AS RANK_DATA
FROM T1
ORDER BY RANK_DATA;
```

DB2 returns the following ranked data:

```
DATA  RANK_DATA
-----
100   1
 35   2
 23   3
  8   4
  8   4
  6   6
```

Suppose that you use the following DENSE_RANK specification on the same data:

```
SELECT DATA,
       DENSE_RANK() OVER (ORDER BY DATA DESC) AS RANK_DATA
FROM T1
ORDER BY RANK_DATA;
```

DB2 returns the following ranked data:

```
DATA  RANK_DATA
-----
100   1
 36   2
 23   3
  8   4
  8   4
  6   5
```

In the example with the RANK specification, two equal values are both ranked as 4. The next rank number is 6. Number 5 is skipped.

In the example with the DENSE_RANK option, those two equal values are also ranked as 4. However, the next rank number is 5. With DENSE_RANK, no gaps exist in the sequential rank numbering.

Related reference:

 OLAP specification (DB2 SQL)

Combining result tables from multiple SELECT statements

When you combine the results of multiple SELECT statements, you can choose what to include in the result table. You can include all rows, only rows that are in the result table of both SELECT statements, or only rows that are unique to the result table of the first SELECT statement.

To combine two or more SELECT statements to form a single result table, use one of the following key words:

UNION

Returns all of the values from the result table of each SELECT statement. If you want all duplicate rows to be repeated in the result table, specify UNION ALL. If you want redundant duplicate rows to be eliminated from the result table, specify UNION or UNION DISTINCT.

EXCEPT

Returns all rows from the first result table (R1) that are not also in the second result table (R2). If you want all duplicate rows from R1 to be contained in the result table, specify EXCEPT ALL. If you want redundant duplicate rows in R1 to be eliminated from the result table, specify EXCEPT or EXCEPT DISTINCT.

INTERSECT

Returns rows that are in the result table of both SELECT statements. If you want all duplicate rows to be contained in the result table, specify INTERSECT ALL. If you want redundant duplicate rows to be eliminated from the result table, specify INTERSECT or INTERSECT DISTINCT.

When you specify one of the preceding set operators (UNION, EXCEPT, or INTERSECT), DB2 processes each SELECT statement to form an interim result table, and then combines the interim result table of each statement. If the nth column of the first result table (R1) and the nth column of the second result table (R2) have the same result column name, the nth column of the result table has that same result column name. If the nth column of R1 and the nth column of R2 do not have the same names, the result column is unnamed.

Examples: Assume that you want to combine the results of two SELECT statements that return the following result tables:

R1 result table

COL1	COL2
a	a
a	b
a	c

R2 result table

```
COL1 COL2
a     b
a     c
a     d
```

A UNION operation combines the two result tables and returns four rows:

```
COL1 COL2
a     a
a     b
a     c
a     d
```

An EXCEPT operation combines the two result tables and returns one row

The result of the EXCEPT operation depends on the which SELECT statement is included before the EXCEPT keyword in the SQL statement. If the SELECT statement that returns the R1 result table is listed first, the result is a single row:

```
COL1 COL2
a     a
```

If the SELECT statement that returns the R2 result table is listed first, the final result is a different row:

```
COL1 COL2
a     d
```

An INTERSECT operation combines the two result tables and returns two rows:

```
COL1 COL2
a     b
a     c
```

Eliminating redundant duplicate rows when combining result tables:

To eliminate redundant duplicate rows when combining result tables, specify one of the following keywords:

- UNION or UNION DISTINCT
- EXCEPT or EXCEPT DISTINCT
- INTERSECT or INTERSECT DISTINCT

To order the entire result table, specify the ORDER BY clause at the end.

Examples: Assume that you have the following tables to manage stock at two book stores.

Table 112. STOCKA

ISBN	TITLE	AUTHOR	NOBEL PRIZE
8778997709	For Whom the Bell Tolls	Hemmingway	N
4599877699	The Good Earth	Buck	Y
9228736278	A Tale of Two Cities	Dickens	N
1002387872	Beloved	Morrison	Y
4599877699	The Good Earth	Buck	Y
0087873532	The Labyrinth of Solitude	Paz	Y

Table 113. STOCKB

ISBN	TITLE	AUTHOR	NOBEL PRIZE
6689038367	The Grapes of Wrath	Steinbeck	Y

Table 113. STOCKB (continued)

ISBN	TITLE	AUTHOR	NOBEL PRIZE
2909788445	The Silent Cry	Oe	Y
1182983745	Light in August	Faulkner	Y
9228736278	A Tale of Two Cities	Dickens	N
1002387872	Beloved	Morrison	Y

Example 1: UNION clause: Suppose that you want a list of books whose authors have won the Nobel Prize and that are in stock at either store. The following SQL statement returns these books in order by author name without redundant duplicate rows:

```
SELECT TITLE, AUTHOR
  FROM STOCKA
 WHERE NOBELPRIZE = 'Y'
UNION
SELECT TITLE, AUTHOR
  FROM STOCKB
 WHERE NOBELPRIZE = 'Y'
ORDER BY AUTHOR
```

This statement returns the following final result table:

Table 114. Result of UNION

TITLE	AUTHOR
The Good Earth	Buck
Light in August	Faulkner
Beloved	Morrison
The Silent Cry	Oe
The Labyrinth of Solitude	Paz
The Grapes of Wrath	Steinbeck

Example 2: EXCEPT: Suppose that you want a list of books that are only in STOCKA. The following SQL statement returns the book names that are in STOCKA only without any redundant duplicate rows:

```
SELECT TITLE
  FROM STOCKA
EXCEPT
SELECT TITLE
  FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

Table 115. Result of EXCEPT

TITLE
For Whom the Bell Tolls
The Good Earth
The Labyrinth of Solitude

Example 3: INTERSECT: Suppose that you want a list of books that are in both STOCKA and in STOCKB. The following statement returns a list of all books from both of these tables with redundant duplicate rows are removed.

```
SELECT TITLE
  FROM STOCKA
INTERSECT
SELECT TITLE
  FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

Table 116. Result of INTERSECT

TITLE
A Tale of Two Cities
Beloved

Keeping all duplicate rows when combining result tables:

To keep all duplicate rows when combining result tables, specify ALL with one of the following set operator keywords:

- UNION ALL
- EXCEPT ALL
- INTERSECT ALL

To order the entire result table, specify the ORDER BY clause at the end.

Examples: The following examples use the STOCKA and STOCK B tables.

Example: UNION ALL: The following SQL statement returns a list of books that won Nobel prizes and are in stock at either store, with duplicates included.

```
SELECT TITLE, AUTHOR
  FROM STOCKA
 WHERE NOBELPRIZE = 'Y'
UNION ALL
SELECT TITLE, AUTHOR
  FROM STOCKB
 WHERE NOBELPRIZE = 'Y'
ORDER BY AUTHOR
```

This statement returns the following result table:

Table 117. Result of UNION ALL

TITLE	AUTHOR
The Good Earth	Buck
The Good Earth	Buck
Light in August	Faulkner
Beloved	Morrison
Beloved	Morrison
The Silent Cry	Oe
The Labyrinth of Solitude	Paz
The Grapes of Wrath	Steinbeck

Example: EXCEPT ALL: Suppose that you want a list of books that are only in STOCKA. The following SQL statement returns the book names that are in STOCKA only with all duplicate rows:

```
SELECT TITLE
  FROM STOCKA
EXCEPT ALL
SELECT TITLE
  FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

Table 118. Result of EXCEPT ALL

TITLE
For Whom the Bell Tolls
The Good Earth
The Good Earth
The Labyrinth of Solitude

Example: INTERSECT ALL clause: Suppose that you want a list of books that are in both STOCKA and in STOCKB, including any duplicate matches. The following statement returns a list of titles that are in both stocks, including duplicate matches. In this case, one match exists for "A Tale of Two Cities" and one match exists for "Beloved."

```
SELECT TITLE
  FROM STOCKA
INTERSECT ALL
SELECT TITLE
  FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

Table 119. Result of INTERSECT ALL

TITLE
A Tale of Two Cities
Beloved

Summarizing group values

You can group rows in the result table by the values of one or more columns or by the results of an expression. You can then apply aggregate functions to each group.

To summarize group values, use GROUP BY.

Except for the columns that are named in the GROUP BY clause, the SELECT statement must specify any other selected columns as an operand of one of the aggregate functions.

Example: GROUP BY clause using one column: The following SQL statement lists, for each department, the lowest and highest education level within that department:

```
SELECT WORKDEPT, MIN(EDLEVEL), MAX(EDLEVEL)
FROM DSN8A10.EMP
GROUP BY WORKDEPT;
```

If a column that you specify in the GROUP BY clause contains null values, DB2 considers those null values to be equal. Thus, all nulls form a single group.

When it is used, the GROUP BY clause follows the FROM clause and any WHERE clause, and it precedes the ORDER BY clause.

You can group the rows by the values of more than one column.

Example: GROUP BY clause using more than one column: The following statement finds the average salary for men and women in departments A00 and C01:

```
SELECT WORKDEPT, SEX, AVG(SALARY) AS AVG_SALARY
FROM DSN8A10.EMP
WHERE WORKDEPT IN ('A00', 'C01')
GROUP BY WORKDEPT, SEX;
```

The result table looks similar to the following output:

WORKDEPT	SEX	AVG_SALARY
A00	F	49625.00000000
A00	M	35000.00000000
C01	F	29722.50000000

DB2 groups the rows first by department number and then (within each department) by sex before it derives the average SALARY value for each group.

You can also group the rows by the results of an expression

Example: GROUP BY clause using a expression: The following statement groups departments by their leading characters, and lists the lowest and highest education level for each group:

```
SELECT SUBSTR(WORKDEPT,1,1), MIN(EDLEVEL), MAX(EDLEVEL)
FROM DSN8A10.EMP
GROUP BY SUBSTR(WORKDEPT,1,1);
```

Filtering groups

If you group rows in the result table, you can also specify a search condition that each retrieved group must satisfy. The search condition tests properties of each group rather than properties of individual rows in the group.

To filter groups, use the HAVING clause to specify a search condition. The HAVING clause acts like a WHERE clause for groups, and it contains the same kind of search conditions that you specify in a WHERE clause.

Example: HAVING clause: The following SQL statement includes a HAVING clause that specifies a search condition for groups of work departments in the employee table:

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY
FROM DSN8A10.EMP
GROUP BY WORKDEPT
HAVING COUNT(*) > 1
ORDER BY WORKDEPT;
```

The result table looks similar to the following output:

WORKDEPT	AVG_SALARY
A00	40850.00000000
C01	29722.50000000
D11	25147.27272727
D21	25668.57142857
E11	21020.00000000
E21	24086.66666666

Compare the preceding example with the second example shown in “Summarizing group values” on page 692. The clause, `HAVING COUNT(*) > 1`, ensures that only departments with more than one member are displayed. In this case, departments B01 and E01 do not display because the `HAVING` clause tests a property of the group.

Example: HAVING clause used with a GROUP BY clause: Use the `HAVING` clause to retrieve the average salary and minimum education level of women in each department for which all female employees have an education level greater than or equal to 16. Assuming that you want results from only departments A00 and D11, the following SQL statement tests the group property, `MIN(EDLEVEL)`:

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY,
       MIN(EDLEVEL) AS MIN_EDLEVEL
FROM DSN8A10.EMP
WHERE SEX = 'F' AND WORKDEPT IN ('A00', 'D11')
GROUP BY WORKDEPT
HAVING MIN(EDLEVEL) >= 16;
```

The result table looks similar to the following output:

WORKDEPT	AVG_SALARY	MIN_EDLEVEL
A00	49625.00000000	18
D11	25817.50000000	17

When you specify both `GROUP BY` and `HAVING`, the `HAVING` clause must follow the `GROUP BY` clause. A function in a `HAVING` clause can include `DISTINCT` if you have not used `DISTINCT` anywhere else in the same `SELECT` statement. You can also connect multiple predicates in a `HAVING` clause with `AND` or `OR`, and you can use `NOT` for any predicate of a search condition.

Finding rows that were changed within a specified period of time

You can filter rows based on the time that they were updated. For example, you might want to find all rows in a particular table that have been changed in the last 7 days.

To find the rows that were changed within a specified period of time, specify the `ROW CHANGE TIMESTAMP` expression in the predicate of your SQL statement.

Recommendation: Ensure that the table has a `ROW CHANGE TIMESTAMP` column that was defined prior to the time period that you want to query. This column ensures that DB2 returns only those rows that were updated in the given time period.

If the table does not have a `ROW CHANGE TIMESTAMP` column, DB2 returns all rows on each page that has had any changes within the given time period. In this case, your result set can contain rows that have not been updated in the give time period, if other rows on that page have been updated or inserted.

Example: Suppose that the TAB table has a ROW CHANGE TIMESTAMP column and that you want to return all of the records that have changed in the last 30 days. The following query returns all of those rows.

```
SELECT * FROM TAB
WHERE ROW CHANGE TIMESTAMP FOR TAB <= CURRENT TIMESTAMP AND
ROW CHANGE TIMESTAMP FOR TAB >= CURRENT TIMESTAMP - 30 days;
```

Example: Suppose that you want to return all of the records that have changed since 9:00 AM January 1, 2004. The following query returns all of those rows.

```
SELECT * FROM TAB
WHERE ROW CHANGE TIMESTAMP FOR TAB >= '2004-01-01-09.00.00';
```

Related reference:

 ROW CHANGE expression (DB2 SQL)

 CREATE TABLE (DB2 SQL)

Joining data from more than one table

Sometimes the information that you want to see is not in a single table. To form a row of the result table, you might want to retrieve some column values from one table and some column values from another table.

You can use a SELECT statement to retrieve and join column values from two or more tables into a single row.

A join operation typically matches a row of one table with a row of another on the basis of a join condition. DB2 supports the following types of joins: inner join, left outer join, right outer join, and full outer join. You can specify joins in the FROM clause of a query.

Nested table expressions and user-defined table functions in joins:

An operand of a join can be more complex than the name of a single table. You can specify one of the following items as a join operand:

nested table expression

A fullselect that is enclosed in parentheses and followed by a correlation name. The correlation name lets you refer to the result of that expression.

Using a nested table expression in a join can be helpful when you want to create a temporary table to use in a join. You can specify the nested table expression as either the right or left operand of a join, depending on which unmatched rows you want included.

user-defined table function

A user-defined function that returns a table.

Using a nested table expression in a join can be helpful when you want to perform some operation on the values in a table before you join them to another table.

Example of using correlated references: In the following SELECT statement, the correlation name that is used for the nested table expression is CHEAP_PARTS. You can use this correlation name to refer to the columns that are returned by the expression. In this case, those correlated references are CHEAP_PARTS.PROD# and CHEAP_PARTS.PRODUCT.

```

SELECT CHEAP_PARTS.PROD#, CHEAP_PARTS.PRODUCT
FROM (SELECT PROD#, PRODUCT
      FROM PRODUCTS
      WHERE PRICE < 10) AS CHEAP_PARTS;

```

The result table looks similar to the following output:

```

PROD#      PRODUCT
=====
505        SCREWDRIVER
30         RELAY

```

The correlated references are valid because they do not occur in the table expression where CHEAP_PARTS is defined. The correlated references are from a table specification at a higher level in the hierarchy of subqueries.

Example of using a nested table expression as the right operand of a join: The following query contains a fullselect (in bold) as the right operand of a left outer join with the PROJECTS table. The correlation name is TEMP. In this case the unmatched rows from the PROJECTS table are included, but the unmatched rows from the nested table expression are not.

```

SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
               PRODUCT, PART, UNITS
FROM PROJECTS LEFT JOIN
  (SELECT PART,
   COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
   PRODUCTS.PRODUCT
  FROM PARTS FULL OUTER JOIN PRODUCTS
   ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP ON PROJECTS.PROD# = PRODNUM;

```

Example of using a nested table expression as the left operand of a join: The following query contains a fullselect as the left operand of a left outer join with the PRODUCTS table. The correlation name is PARTX. In this case the unmatched rows from the nested table expression are included, but the unmatched rows from the PRODUCTS table are not.

```

SELECT PART, SUPPLIER, PRODNUM, PRODUCT
FROM (SELECT PART, PROD# AS PRODNUM, SUPPLIER
      FROM PARTS
      WHERE PROD# < '200') AS PARTX
LEFT OUTER JOIN PRODUCTS
ON PRODNUM = PROD#;

```

The result table looks similar to the following output:

```

PART      SUPPLIER      PRODNUM      PRODUCT
=====
WIRE      ACWF          10           GENERATOR
MAGNETS   BATEMAN      10           GENERATOR
OIL       WESTERN_CHEM 160          -----

```

Because PROD# is a character field, DB2 does a character comparison to determine the set of rows in the result. Therefore, because the characters '30' are greater than '200', the row in which PROD# is equal to '30' does not appear in the result.

Example: Using a table function as an operand of a join: Suppose that CVTPRICE is a table function that converts the prices in the PRODUCTS table to the currency that you specify and returns the PRODUCTS table with the prices in those units. You can obtain a table of parts, suppliers, and product prices with the prices in your choice of currency by executing a query similar to the following query:

```

SELECT PART, SUPPLIER, PARTS.PROD#, Z.PRODUCT, Z.PRICE
FROM PARTS, TABLE(CVTPRICE(:CURRENCY)) AS Z
WHERE PARTS.PROD# = Z.PROD#;

```

Correlated references in table specifications in joins:

Use correlation names to refer to the results of a nested table expression. After you specify the correlation name for an expression, any subsequent reference to this correlation name is called a *correlated reference*.

You can include correlated references in nested table expressions or as arguments to table functions. The basic rule that applies for both of these cases is that the correlated reference must be from a table specification at a higher level in the hierarchy of subqueries. You can also use a correlated reference and the table specification to which it refers in the same FROM clause if the table specification appears to the left of the correlated reference and the correlated reference is in one of the following clauses:

- A nested table expression that is preceded by the keyword TABLE
- The argument of a table function

For more information about correlated references, see “Correlation names in references” on page 716.

A table function or a table expression that contains correlated references to other tables in the same FROM clause cannot participate in a full outer join or a right outer join. The following examples illustrate valid uses of correlated references in table specifications.

Example: In this example, the correlated reference T.C2 is valid because the table specification, to which it refers, T, is to its left.

```

SELECT T.C1, Z.C5
FROM T, TABLE(TF3(T.C2)) AS Z
WHERE T.C3 = Z.C4;

```

If you specify the join in the opposite order, with T following TABLE(TF3(T.C2)), T.C2 is invalid.

Example: In this example, the correlated reference D.DEPTNO is valid because the nested table expression within which it appears is preceded by TABLE, and the table specification D appears to the left of the nested table expression in the FROM clause.

```

SELECT D.DEPTNO, D.DEPTNAME,
EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
FROM DEPT D,
TABLE(SELECT AVG(E.SALARY) AS AVGSAL,
COUNT(*) AS EMPCOUNT
FROM EMP E
WHERE E.WORKDEPT=D.DEPTNO) AS EMPINFO;

```

If you remove the keyword TABLE, D.DEPTNO is invalid.

Joining more than two tables

Joins are not limited to two tables. You can join more than two tables in a single SQL statement.

To join more than two tables, specify join conditions that include columns from all of the relevant tables.

Example: Suppose that you want a result table that shows employees who have projects that they are responsible for, their projects, and their department names. You need to join three tables to get all the information. You can use the following SELECT statement:

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
      FROM DSN8A10.EMP, DSN8A10.PROJ, DSN8A10.DEPT
      WHERE EMPNO = RESPEMP
      AND WORKDEPT = DSN8A10.DEPT.DEPTNO;
```

The result table looks similar to the following output:

EMPNO	LASTNAME	DEPTNAME	PROJNO
000010	HAAS	SPIFFY COMPUTER SERVICE DIV	AD3100
000010	HAAS	SPIFFY COMPUTER SERVICE DIV	MA2100
000020	THOMPSON	PLANNING	PL2100
000030	KWAN	INFORMATION CENTER	IF1000
000030	KWAN	INFORMATION CENTER	IF2000
000050	GEYER	SUPPORT SERVICES	OP1000
000050	GEYER	SUPPORT SERVICES	OP2000
000060	STERN	MANUFACTURING SYSTEMS	MA2110
000070	PULASKI	ADMINISTRATION SYSTEMS	AD3110
000090	HENDERSON	OPERATIONS	OP1010
000100	SPENSER	SOFTWARE SUPPORT	OP2010
000150	ADAMSON	MANUFACTURING SYSTEMS	MA2112
000160	PIANKA	MANUFACTURING SYSTEMS	MA2113
000220	LUTZ	MANUFACTURING SYSTEMS	MA2111
000230	JEFFERSON	ADMINISTRATION SYSTEMS	AD3111
000250	SMITH	ADMINISTRATION SYSTEMS	AD3112
000270	PEREZ	ADMINISTRATION SYSTEMS	AD3113
000320	MEHTA	SOFTWARE SUPPORT	OP2011
000330	LEE	SOFTWARE SUPPORT	OP2012
000340	GOUNOT	SOFTWARE SUPPORT	OP2013

DB2 determines the intermediate and final results of the previous query by performing the following logical steps:

1. Join the employee and project tables on the employee number, dropping the rows with no matching employee number in the project table.
2. Join the intermediate result table with the department table on matching department numbers.
3. Process the select list in the final result table, leaving only four columns.

Joining more than two tables by using more than one join type:

When joining more than two tables, you do not have to use the same join type for every join.

To join tables by using more than one join type, specify the join types in the FROM clause.

Example: Suppose that you want a result table that shows the following items:

- employees whose last name begins with 'S' or a letter that comes after 'S' in the alphabet
- the department names for these employees
- any projects that these employees are responsible for

You can use the following SELECT statement:

```

SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM DSN8A10.EMP INNER JOIN DSN8A10.DEPT
    ON WORKDEPT = DSN8A10.DEPT.DEPTNO
  LEFT OUTER JOIN DSN8A10.PROJ
    ON EMPNO = RESPEMP
 WHERE LASTNAME > 'S';

```

The result table looks like similar to the following output:

EMPNO	LASTNAME	DEPTNAME	PROJNO
000020	THOMPSON	PLANNING	PL2100
000060	STERN	MANUFACTURING SYSTEMS	MA2110
000100	SPENSER	SOFTWARE SUPPORT	OP2010
000170	YOSHIMURA	MANUFACTURING SYSTEMS	-----
000180	SCOUTTEN	MANUFACTURING SYSTEMS	-----
000190	WALKER	MANUFACTURING SYSTEMS	-----
000250	SMITH	ADMINISTRATION SYSTEMS	AD3112
000280	SCHNEIDER	OPERATIONS	-----
000300	SMITH	OPERATIONS	-----
000310	SETRIGHT	OPERATIONS	-----
200170	YAMAMOTO	MANUFACTURING SYSTEMS	-----
200280	SCHWARTZ	OPERATIONS	-----
200310	SPRINGER	OPERATIONS	-----
200330	WONG	SOFTWARE SUPPORT	-----

DB2 determines the intermediate and final results of the previous query by performing the following logical steps:

1. Join the employee and department tables on matching department numbers, dropping the rows where the last name begins with a letter before 'S in the alphabet'.
2. Join the intermediate result table with the project table on the employee number, keeping the rows for which no matching employee number exists in the project table.
3. Process the select list in the final result table, leaving only four columns.

Inner joins

An *inner join* is a method of combining two tables that discards rows of either table that do not match any row of the other table. The matching is based on the join condition.

To request an inner join, execute a SELECT statement in which you specify the tables that you want to join in the FROM clause, and specify a WHERE clause or an ON clause to indicate the join condition. The join condition can be any simple or compound search condition that does not contain a subquery reference.

In the simplest type of inner join, the join condition is *column1=column2*.

Example

You can join the PARTS and PRODUCTS tables in sample data from joins on the PROD# column to get a table of parts with their suppliers and the products that use the parts.

To do this, you can use either one of the following SELECT statements:

```

SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS, PRODUCTS
 WHERE PARTS.PROD# = PRODUCTS.PROD#;

SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;

```

The result table looks like the following output:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW

Three things about this example:

- A part in the parts table (OIL) has product (#160), which is not in the products table. A product (SCREWDRIVER, #505) has no parts listed in the parts table. Neither OIL nor SCREWDRIVER appears in the result of the join.
In contrast, an *outer join* includes rows in which the values in the joined columns do not match.
- You can explicitly specify that this join is an inner join (not an outer join). Use INNER JOIN in the FROM clause instead of the comma, and use ON to specify the join condition (rather than WHERE) when you explicitly join tables in the FROM clause.
- If you do not specify a WHERE clause in the first form of the query, the result table contains all possible combinations of rows for the tables that are identified in the FROM clause. You can obtain the same result by specifying a join condition that is always true in the second form of the query, as in the following statement:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON 1=1;
```

Regardless of whether you omit the WHERE clause or specify a join condition that is always true, the number of rows in the result table is the product of the number of rows in each table.

You can specify more complicated join conditions to obtain different sets of results. For example, to eliminate the suppliers that begin with the letter A from the table of parts, suppliers, product numbers, and products, write a query like the following query:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#
AND SUPPLIER NOT LIKE 'A%';
```

The result of the query is all rows that do not have a supplier that begins with A. The result table looks like the following output:

PART	SUPPLIER	PROD#	PRODUCT
=====	=====	=====	=====
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY

Example of joining a table to itself by using an inner join

Joining a table to itself is useful to show relationships between rows. The following example returns a list of major projects from the PROJ table and the projects that are part of those major projects.

In this example, **A** indicates the first instance of table DSN8A10.PROJ, and **B** indicates the second instance of this table. The join condition is such that the value in column PROJNO in table DSN8A10.PROJ A must be equal to a value in column MAJPROJ in table DSN8A10.PROJ B.

The following SQL statement joins table DSN8A10.PROJ to itself and returns the number and name of each major project followed by the number and name of the project that is part of it:

```
SELECT A.PROJNO, A.PROJNAME, B.PROJNO, B.PROJNAME
FROM DSN8A10.PROJ A, DSN8A10.PROJ B
WHERE A.PROJNO = B.MAJPROJ;
```

The result table looks similar to the following output:

PROJNO	PROJNAME	PROJNO	PROJNAME
AD3100	ADMIN SERVICES	AD3110	GENERAL AD SYSTEMS
AD3110	GENERAL AD SYSTEMS	AD3111	PAYROLL PROGRAMMING
AD3110	GENERAL AD SYSTEMS	AD3112	PERSONNEL PROGRAMMG
:	:	:	:
OP2010	SYSTEMS SUPPORT	OP2013	DB/DC SUPPORT

In this example, the comma in the FROM clause implicitly specifies an inner join, and it acts the same as if the INNER JOIN keywords had been used. When you use the comma for an inner join, you must specify the join condition on the WHERE clause. When you use the INNER JOIN keywords, you must specify the join condition on the ON clause.

Related reference:

“Sample data for joins” on page 706

 from-clause (DB2 SQL)

Outer joins

An *outer join* is a method of combining two or more tables so that the result includes unmatched rows of one of the tables, or of both tables. The matching is based on the join condition.

DB2 supports three types of outer joins:

full outer join

Includes unmatched rows from both tables. If any column of the result table does not have a value, that column has the null value in the result table.

left outer join

Includes rows from the table that is specified before LEFT OUTER JOIN that have no matching values in the table that is specified after LEFT OUTER JOIN.

right outer join

Includes rows from the table that is specified after RIGHT OUTER JOIN that have no matching values in the table that is specified before RIGHT OUTER JOIN.

The following table illustrates how the PARTS and PRODUCTS tables in “Sample data for joins” on page 706 can be combined using the three outer join functions.

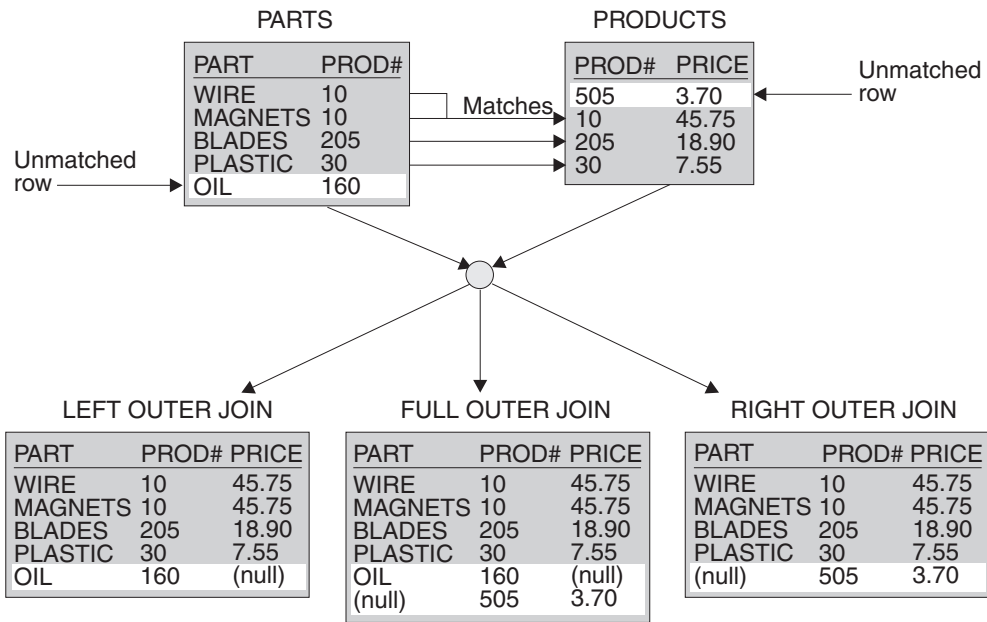


Figure 37. Three outer joins from the PARTS and PRODUCTS tables

The result table contains data that is joined from all of the tables, for rows that satisfy the search conditions.

The result columns of a join have names if the outermost SELECT list refers to base columns. However, if you use a function (such as COALESCE or VALUE) to build a column of the result, that column does not have a name unless you use the AS clause in the SELECT list.

Full outer join

An *full outer join* is a method of combining tables so that the result includes unmatched rows of both tables.

If you are joining two tables and want the result set to include unmatched rows from both tables, use a FULL OUTER JOIN clause. The matching is based on the join condition. If any column of the result table does not have a value, that column has the null value in the result table.

The join condition for a full outer join must be a simple search condition that compares two columns or an invocation of a cast function that has a column name as its argument.

Example: The following query performs a full outer join of the PARTS and PRODUCTS tables in “Sample data for joins” on page 706:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result table from the query looks similar to the following output:

PART	SUPPLIER	PROD#	PRODUCT
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY

BLADES	ACE_STEEL	205	SAW
OIL	WESTERN_CHEM	160	-----
-----	-----	---	SCREWDRIVER

Example of using COALESCE or VALUE: COALESCE is the keyword that is specified by the SQL standard as a synonym for the VALUE function. This function, by either name, can be particularly useful in full outer join operations because it returns the first non-null value from the pair of join columns.

The product number in the result of the example for “Full outer join” on page 702 is null for SCREWDRIVER, even though the PRODUCTS table contains a product number for SCREWDRIVER. If you select PRODUCTS.PROD# instead, PROD# is null for OIL. If you select both PRODUCTS.PROD# and PARTS.PROD#, the result contains two columns, both of which contain some null values. You can merge data from both columns into a single column, eliminating the null values, by using the COALESCE function.

With the same PARTS and PRODUCTS tables, the following example merges the non-null data from the PROD# columns:

```
SELECT PART, SUPPLIER,
       COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result table looks similar to the following output:

PART	SUPPLIER	PRODNUM	PRODUCT
-----	-----	-----	-----
WIRE	ACWF	10	GENERATOR
MAGNETS	BATEMAN	10	GENERATOR
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW
OIL	WESTERN_CHEM	160	-----
-----	-----	505	SCREWDRIVER

The AS clause (AS PRODNUM) provides a name for the result of the COALESCE function.

Left outer join

A *left outer join* is a method of combining tables. The result includes unmatched rows from only the table that is specified before the LEFT OUTER JOIN clause.

If you are joining two tables and want the result set to include unmatched rows from only one table, use a LEFT OUTER JOIN clause or a RIGHT OUTER JOIN clause. The matching is based on the join condition.

The clause LEFT OUTER JOIN includes rows from the table that is specified before LEFT OUTER JOIN that have no matching values in the table that is specified after LEFT OUTER JOIN.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

Example: The following example uses the tables in “Sample data for joins” on page 706. To include rows from the PARTS table that have no matching values in the PRODUCTS table, and to include prices that exceed 10.00, run the following query:

```

SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT, PRICE
  FROM PARTS LEFT OUTER JOIN PRODUCTS
    ON PARTS.PROD#=PRODUCTS.PROD#
   AND PRODUCTS.PRICE>10.00;

```

The result table looks similar to the following output:

PART	SUPPLIER	PROD#	PRODUCT	PRICE
WIRE	ACWF	10	GENERATOR	45.75
MAGNETS	BATEMAN	10	GENERATOR	45.75
PLASTIC	PLASTIK_CORP	30		
BLADES	ACE_STEEL	205	SAW	18.90
OIL	WESTERN_CHEM	160		

A row from the PRODUCTS table is in the result table only if its product number matches the product number of a row in the PARTS table and the price is greater than 10.00 for that row. Rows in which the PRICE value does not exceed 10.00 are included in the result of the join, but the PRICE value is set to null.

In this result table, the row for PROD# 30 has null values on the right two columns because the price of PROD# 30 is less than 10.00. PROD# 160 has null values on the right two columns because PROD# 160 does not match another product number.

Right outer join

A *right outer join* is a method of combining tables. The result includes unmatched rows from only the table that is specified after the RIGHT OUTER JOIN clause.

If you are joining two tables and want the result set to include unmatched rows from only one table, use a LEFT OUTER JOIN clause or a RIGHT OUTER JOIN clause. The matching is based on the join condition.

The clause RIGHT OUTER JOIN includes rows from the table that is specified after RIGHT OUTER JOIN that have no matching values in the table that is specified before RIGHT OUTER JOIN.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

Example: The following example uses the tables in “Sample data for joins” on page 706. To include rows from the PRODUCTS table that have no corresponding rows in the PARTS table, execute this query:

```

SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT, PRICE
  FROM PARTS RIGHT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
   AND PRODUCTS.PRICE>10.00;

```

The result table looks similar to the following output:

PART	SUPPLIER	PROD#	PRODUCT	PRICE
WIRE	ACWF	10	GENERATOR	45.75
MAGNETS	BATEMAN	10	GENERATOR	45.75
BLADES	ACE_STEEL	205	SAW	18.90
		30	RELAY	7.55
		505	SCREWDRIVER	3.70

A row from the PARTS table is in the result table only if its product number matches the product number of a row in the PRODUCTS table and the price is greater than 10.00 for that row.

Because the PRODUCTS table can have rows with nonmatching product numbers in the result table, and the PRICE column is in the PRODUCTS table, rows in which PRICE is less than or equal to 10.00 are included in the result. The PARTS columns contain null values for these rows in the result table.

SQL rules for statements that contain join operations

Typically, DB2 performs a join operation first, before it evaluates the other clauses of the SELECT statement.

SQL rules dictate that the result of a SELECT statement look as if the clauses had been evaluated in this order:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT

A join operation is part of a FROM clause; therefore, for the purpose of predicting which rows will be returned from a SELECT statement that contains a join operation, assume that the join operation is performed first.

Example: Suppose that you want to obtain a list of part names, supplier names, product numbers, and product names from the PARTS and PRODUCTS tables. You want to include rows from either table where the PROD# value does not match a PROD# value in the other table, which means that you need to do a full outer join. You also want to exclude rows for product number 10. Consider the following SELECT statement:

```
SELECT PART, SUPPLIER,
       VALUE(PARTS.PROD#,PRODUCTS.PROD#) AS PRODNUM, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#
WHERE PARTS.PROD# <> '10' AND PRODUCTS.PROD# <> '10';
```

The following result is **not** what you wanted:

PART	SUPPLIER	PRODNUM	PRODUCT
PLASTIC	PLASTIK_CORP	30	RELAY
BLADES	ACE_STEEL	205	SAW

DB2 performs the join operation first. The result of the join operation includes rows from one table that do not have corresponding rows from the other table. However, the WHERE clause then excludes the rows from both tables that have null values for the PROD# column.

The following statement is a correct SELECT statement to produce the list:

```
SELECT PART, SUPPLIER,
       VALUE(X.PROD#, Y.PROD#) AS PRODNUM, PRODUCT
FROM
  (SELECT PART, SUPPLIER, PROD# FROM PARTS WHERE PROD# <> '10') X
FULL OUTER JOIN
  (SELECT PROD#, PRODUCT FROM PRODUCTS WHERE PROD# <> '10') Y
ON X.PROD# = Y.PROD#;
```

For this statement, DB2 applies the WHERE clause to each table separately. DB2 then performs the full outer join operation, which includes rows in one table that do not have a corresponding row in the other table. The final result includes rows with the null value for the PROD# column and looks similar to the following output:

PART	SUPPLIER	PRODNUM	PRODUCT
=====	=====	=====	=====
OIL	WESTERN_CHEM	160	-----
BLADES	ACE_STEEL	205	SAW
PLASTIC	PLASTIK_CORP	30	RELAY
-----	-----	505	SCREWDRIVER

Sample data for joins

You can use the sample PARTS table and the PRODUCTS table to practice various types of joins.

The examples in these topics use the following two tables to show various types of joins:

<i>The PARTS table</i>			<i>The PRODUCTS table</i>		
PART	PROD#	SUPPLIER	PROD#	PRODUCT	PRICE
=====	=====	=====	=====	=====	=====
WIRE	10	ACWF	505	SCREWDRIVER	3.70
OIL	160	WESTERN_CHEM	30	RELAY	7.55
MAGNETS	10	BATEMAN	205	SAW	18.90
PLASTIC	30	PLASTIK_CORP	10	GENERATOR	45.75
BLADES	205	ACE_STEEL			

Optimizing retrieval for a small set of rows

When you need only a few of the thousands of rows that satisfy a query, you can tell DB2 to optimize its retrieval process to return only a specified number of rows.

Question: How can I tell DB2 that I want only a few of the thousands of rows that satisfy a query?

Answer: Use OPTIMIZE FOR *n* ROWS or FETCH FIRST *n* ROWS ONLY.

DB2 usually optimizes queries to retrieve all rows that qualify. But sometimes you want to retrieve only the first few rows. For example, to retrieve the first row that is greater than or equal to a known value, code:

```
SELECT column list FROM table
WHERE key >= value
ORDER BY key ASC
```

Even with the ORDER BY clause, DB2 might fetch all the data first and sort it afterwards, which could be wasteful. Instead, you can write the query in one of the following ways:

```
SELECT * FROM table
WHERE key >= value
ORDER BY key ASC
OPTIMIZE FOR 1 ROW

SELECT * FROM table
WHERE key >= value
ORDER BY key ASC
FETCH FIRST n ROWS ONLY
```

Use OPTIMIZE FOR 1 ROW to influence the access path. OPTIMIZE FOR 1 ROW tells DB2 to select an access path that returns the first qualifying row quickly.

Use FETCH FIRST *n* ROWS ONLY to limit the number of rows in the result table to *n* rows. FETCH FIRST *n* ROWS ONLY has the following benefits:

- When you use FETCH statements to retrieve data from a result table, FETCH FIRST *n* ROWS ONLY causes DB2 to retrieve only the number of rows that you

need. This can have performance benefits, especially in distributed applications. If you try to execute a FETCH statement to retrieve the $n+1$ st row, DB2 returns a +100 SQLCODE.


- When you use FETCH FIRST ROW ONLY in a SELECT INTO statement, you never retrieve more than one row. Using FETCH FIRST ROW ONLY in a SELECT INTO statement can prevent SQL errors that are caused by inadvertently selecting more than one value into a host variable.

When you specify FETCH FIRST n ROWS ONLY but not OPTIMIZE FOR n ROWS, OPTIMIZE FOR n ROWS is implied. When you specify FETCH FIRST n ROWS ONLY and OPTIMIZE FOR m ROWS, and m is less than n , DB2 optimizes the query for m rows. If m is greater than n , DB2 optimizes the query for n rows.

Related concepts:

 Optimization for large and small result sets (Introduction to DB2 for z/OS)

Related tasks:

 Optimizing retrieval for a small set of rows (DB2 Application programming and SQL)

 Fetching a limited number of rows (DB2 Performance)

Related reference:

 optimize-clause (DB2 SQL)

 fetch-first-clause (DB2 SQL)

Creating recursive SQL by using common table expressions

Queries that use recursion are useful in applications like bill-of-materials applications, network planning applications, and reservation systems.

You can use common table expressions to create recursive SQL. If a fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*.

Recursive common table expressions must follow these rules:

- The first fullselect of the first union (the initialization fullselect) must not include a reference to the common table expression.
- Each fullselect that is part of the recursion cycle must:
 - Start with SELECT or SELECT ALL. SELECT DISTINCT is not allowed.
 - Include only one reference to the common table expression that is part of the recursion cycle in its FROM clause.
 - Not include aggregate functions, a GROUP BY clause, or a HAVING clause.
- The column names must be specified after the table name of the common table expression.
- The data type, length, and CCSID of each column from the common table expression must match the data type, length, and CCSID of each corresponding column in the iterative fullselect.
- If you use the UNION keyword, specify UNION ALL instead of UNION.
- You cannot specify INTERSECT or EXCEPT.
- Outer joins must not be part of any recursion cycle.
- A subquery must not be part of any recursion cycle.

Important: You should be careful to avoid an infinite loop when you use a recursive common table expression. DB2 issues a warning if one of the following items is **not** found in the iterative fullselect of a recursive common table expression:

- An integer column that increments by a constant
- A predicate in the WHERE clause in the form of *counter_column < constant* or *counter_column < :host variable*

See “Examples of recursive common table expressions” on page 489 for examples of bill-of-materials applications that use recursive common table expressions.

Updating data as it is retrieved from the database

As you retrieve rows, you can update them at the same time.

Question: How can I update rows of data as I retrieve them?

Answer: On the SELECT statement, use the FOR UPDATE clause without a column list, or the FOR UPDATE OF clause with a column list. For a more efficient program, specify a column list with only those columns that you intend to update. Then use the positioned UPDATE statement. The clause WHERE CURRENT OF identifies the cursor that points to the row you want to update.

Avoiding decimal arithmetic errors

When you request that DB2 perform a decimal operation, errors might occur if DB2 does not use the appropriate precision and scale.

For static SQL statements, the simplest way to avoid a division error is to override DEC31 rules by specifying the precompiler option DEC(15). In some cases you can avoid a division error by specifying D31.s, where *s* is a number between 1 and 9 and represents the minimum scale to be used for division operations. This specification reduces the probability of errors for statements that are embedded in the program.

If the dynamic SQL statements have bind, define, or invoke behavior and the value of the installation option for USE FOR DYNAMICRULES on panel DSNTIP4 is NO, you can use the precompiler option DEC(15), DEC15, or D15.s to override DEC31 rules, where *s* is a number between 1 and 9.

For a dynamic statement, or for a single static statement, use the scalar function DECIMAL to specify values of the precision and scale for a result that causes no errors.

Before you execute a dynamic statement, set the value of special register CURRENT PRECISION to DEC15 or D15.s, where *s* is a number between 1 and 9.

Even if you use DEC31 rules, multiplication operations can sometimes cause overflow because the precision of the product is greater than 31. To avoid overflow from multiplication of large numbers, use the MULTIPLY_ALT built-in function instead of the multiplication operator.

Precision for operations with decimal numbers

DB2 accepts two sets of rules for determining the precision and scale of the result of an operation with decimal numbers.

- DEC15 rules allow a maximum precision of 15 digits in the result of an operation. DEC15 rules are in effect when both operands have a precision of 15 or less, or unless the DEC31 rules apply.
- DEC31 rules allow a maximum precision of 31 digits in the result. DEC31 rules are in effect if any of the following conditions is true:
 - Either operand of the operation has a precision greater than 15 digits.
 - The operation is in a dynamic SQL statement, and any of the following conditions is true:
 - The current value of special register CURRENT PRECISION is DEC31 or D31.s, where s is a number between 1 and 9 and represents the minimum scale to be used for division operations.
 - The installation option for DECIMAL ARITHMETIC on panel DSNTIP4 is DEC31, 31, or D31.s, where s is a number between 1 and 9; the installation option for USE FOR DYNAMICRULES on panel DSNTIP4 is YES; and the value of CURRENT PRECISION has not been set by the application.
 - The SQL statement has bind, define, or invoke behavior; the statement is in an application that is precompiled with option DEC(31); the installation option for USE FOR DYNAMICRULES on panel DSNTIP4 is NO; and the value of CURRENT PRECISION has not been set by the application. See “DYNAMICRULES bind option” on page 959 for an explanation of bind, define, and invoke behavior.
 - The operation is in an embedded (static) SQL statement that you precompiled with the DEC(31), DEC31, or D31.s option, or with the default for that option when the installation option DECIMAL ARITHMETIC is DEC31 or 31. s is a number between 1 and 9 and represents the minimum scale to be used for division operations. See “Processing SQL statements” on page 918 for information about precompiling and for a list of all precompiler options.

Recommendation: To reduce the chance of overflow, or when dealing with a precision greater than 15 digits, choose DEC31 or D31.s, where s is a number between 1 and 9 and represents the minimum scale to be used for division operations.

Controlling how DB2 rounds decimal floating point numbers

You can specify a default rounding mode that DB2 is to use for all DECFLOAT values.

To control how DB2 rounds decimal floating point numbers:

Set the CURRENT DECFLOAT ROUNDING MODE special register.

Related reference:

 CURRENT DECFLOAT ROUNDING MODE (DB2 SQL)

 SET CURRENT DECFLOAT ROUNDING MODE (DB2 SQL)

Implications of using SELECT *

Generally, you should use SELECT * only when you want to select all columns, except for hidden columns. Otherwise, specify the specific columns that you want to view.

Question: What are the implications of using SELECT * ?

Answer: Generally, you should select only the columns you need because DB2 is sensitive to the number of columns selected. Use SELECT * only when you are sure you want to select all columns, except hidden columns. (Hidden columns are not returned when you specify SELECT *.) One alternative to selecting all columns is to use views defined with only the necessary columns, and use SELECT * to access the views. Avoid SELECT * if all the selected columns participate in a sort operation (SELECT DISTINCT and SELECT...UNION, for example).

Subqueries

When you need to narrow your search condition based on information in an interim table, you can use a subquery. For example, you might want to find all employee numbers in one table that also exist for a given project in a second table.

Conceptual overview of subqueries

Suppose that you want a list of the employee numbers, names, and commissions of all employees who work on a particular project, whose project number is MA2111. The first part of the SELECT statement is easy to write:

```
SELECT EMPNO, LASTNAME, COMM
      FROM DSN8A10.EMP
      WHERE EMPNO
```

:

However, you cannot proceed because the DSN8A10.EMP table does not include project number data. You do not know which employees are working on project MA2111 without issuing another SELECT statement against the DSN8A10.EMPPROJECT table.

You can use a subquery to solve this problem. A *subquery* is a subselect or a fullselect in a WHERE clause. The SELECT statement that surrounds the subquery is called the *outer SELECT*.

```
SELECT EMPNO, LASTNAME, COMM
      FROM DSN8A10.EMP
      WHERE EMPNO IN
            (SELECT EMPNO
             FROM DSN8A10.EMPPROJECT
             WHERE PROJNO = 'MA2111');
```

To better understand the results of this SQL statement, imagine that DB2 goes through the following process:

1. DB2 evaluates the subquery to obtain a list of EMPNO values:

```
(SELECT EMPNO
 FROM DSN8A10.EMPPROJECT
 WHERE PROJNO = 'MA2111');
```

The result is in an interim result table, similar to the one in the following output:

```
from EMPNO
=====
200
200
220
```

2. The interim result table then serves as a list in the search condition of the outer SELECT. Effectively, DB2 executes this statement:


```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8A10.EMP
WHERE EMPNO IN
('000200', '000220');
```

As a consequence, the result table looks similar to the following output:

EMPNO	LASTNAME	COMM
000200	BROWN	2217
000220	LUTZ	2387

Correlated and uncorrelated subqueries

Subqueries supply information that is needed to qualify a row (in a WHERE clause) or a group of rows (in a HAVING clause). The subquery produces a result table that is used to qualify the row or group of selected rows.

A subquery executes only once, if the subquery is the same for every row or group. This kind of subquery is *uncorrelated*, which means that it executes only once. For example, in the following statement, the content of the subquery is the same for every row of the table DSN8A10.EMP:

```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8A10.EMP
WHERE EMPNO IN
(SELECT EMPNO
FROM DSN8A10.EMPPROJACT
WHERE PROJNO = 'MA2111');
```

Subqueries that vary in content from row to row or group to group are *correlated* subqueries. For information about correlated subqueries, see “Correlated subqueries” on page 714.

Subqueries and predicates

A *predicate* is an element of a search condition that specifies a condition that is true, false, or unknown about a given row or group. A subquery, which is a SELECT statement within the WHERE or HAVING clause of another SQL statement, is always part of a predicate. The predicate is of the form:

operand operator (subquery)

A WHERE or HAVING clause can include predicates that contain subqueries. A predicate that contains a subquery, like any other search predicate, can be enclosed in parentheses, can be preceded by the keyword NOT, and can be linked to other predicates through the keywords AND and OR. For example, the WHERE clause of a query can look something like the following clause:

```
WHERE X IN (subquery1) AND (Y > SOME (subquery2)) OR Z IS NULL
```

Subqueries can also appear in the predicates of other subqueries. Such subqueries are nested subqueries at some level of nesting. For example, a subquery within a subquery within an outer SELECT has a nesting level of 2. DB2 allows nesting down to a level of 15, but few queries require a nesting level greater than 1.

The relationship of a subquery to its outer SELECT is the same as the relationship of a nested subquery to a subquery, and the same rules apply, except where otherwise noted.

The subquery result table

A subquery must produce a result table that has the same number of columns as the number of columns on the left side of the comparison operator. For example, both of the following SELECT statements are acceptable:

```
SELECT EMPNO, LASTNAME
  FROM DSN8A10.EMP
 WHERE SALARY =
   (SELECT AVG(SALARY)
    FROM DSN8A10.EMP);

SELECT EMPNO, LASTNAME
  FROM DSN8A10.EMP
 WHERE (SALARY, BONUS) IN
   (SELECT AVG(SALARY), AVG(BONUS)
    FROM DSN8A10.EMP);
```

Except for a subquery of a basic predicate, the result table can contain more than one row. For more information, see “Places where you can include a subquery.”

Related concepts:

- [Subquery access \(DB2 Performance\)](#)
- [Predicates \(DB2 SQL\)](#)

Related tasks:

- [Writing efficient subqueries \(DB2 Performance\)](#)

Related reference:

- [where-clause \(DB2 SQL\)](#)
- [having-clause \(DB2 SQL\)](#)

Places where you can include a subquery

You can specify a subquery in either a WHERE clause or a HAVING clause.

You can specify a subquery in either a WHERE or HAVING clause by using one of the following items:

Example: Basic predicate in a subquery

You can use a subquery immediately after any of the comparison operators. If you do, the subquery can return at most one value. DB2 compares that value with the value to the left of the comparison operator.

The following SQL statement returns the employee numbers, names, and salaries for employees whose education level is higher than the average company-wide education level.

```
SELECT EMPNO, LASTNAME, SALARY
  FROM DSN8A10.EMP
 WHERE EDLEVEL >
   (SELECT AVG(EDLEVEL)
    FROM DSN8A10.EMP);
```

Example: Quantified predicate in a subquery: ALL, ANY, or SOME

You can use a subquery after a comparison operator, followed by the keyword ALL, ANY, or SOME. The number of columns and rows that the subquery can return for a quantified predicate depends on the type of quantified predicate:

- For = SOME, = ANY, or <> ALL, the subquery can return one or many rows and one or many columns. The number of columns in the result table must match the number of columns on the left side of the operator.
- For all other quantified predicates, the subquery can return one or many rows, but no more than one column.

See the information about quantified predicates, including what to do if a subquery that returns one or more null values gives you unexpected results.

Example: ALL predicate

Use ALL to indicate that the operands on the left side of the comparison must compare in the same way with **all** of the values that the subquery returns. For example, suppose that you use the greater-than comparison operator with ALL:

```
WHERE column > ALL (subquery)
```

To satisfy this WHERE clause, the column value must be greater than all of the values that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

Now suppose that you use the <> operator with ALL in a WHERE clause like this:

```
WHERE (column1, column1, ... columnn) <> ALL (subquery)
```

To satisfy this WHERE clause, each column value must be unequal to all of the values in the corresponding column of the result table that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

Example: ANY or SOME predicate

Use ANY or SOME to indicate that the values on the left side of the operator must compare in the indicated way to **at least one** of the values that the subquery returns. For example, suppose that you use the greater-than comparison operator with ANY:

```
WHERE expression > ANY (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than at least one of the values (that is, greater than the lowest value) that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

Now suppose that you use the = operator with SOME in a WHERE clause like this:

```
WHERE (column1, column1, ... columnn) = SOME (subquery)
```

To satisfy this WHERE clause, each column value must be equal to at least one of the values in the corresponding column of the result table that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

Example: IN predicate in a subquery

You can use IN to say that the value or values on the left side of the IN operator must be among the values that are returned by the subquery. Using IN is equivalent to using = ANY or = SOME.

The following query returns the names of department managers:

```
SELECT EMPNO, LASTNAME
FROM DSN8A10.EMP
WHERE EMPNO IN
  (SELECT DISTINCT MGRNO
   FROM DSN8A10.DEPT);
```

EXISTS predicate in a subquery

When you use the keyword EXISTS, DB2 checks whether the subquery returns one or more rows. Returning one or more rows satisfies the condition; returning no rows does not satisfy the condition.

The search condition in the following query is satisfied if any project that is represented in the project table has an estimated start date that is later than 1 January 2005:

```
SELECT EMPNO, LASTNAME
FROM DSN8A10.EMP
WHERE EXISTS
  (SELECT *
   FROM DSN8A10.PROJ
   WHERE PRSTDATE > '2005-01-01');
```

The result of the subquery is always the same for every row that is examined for the outer SELECT. Therefore, either every row appears in the result of the outer SELECT or none appears. A correlated subquery is more powerful than the uncorrelated subquery that is used in this example because the result of a correlated subquery is evaluated for each row of the outer SELECT.

As shown in the example, you do not need to specify column names in the subquery of an EXISTS clause. Instead, you can code SELECT *. You can also use the EXISTS keyword with the NOT keyword in order to select rows when the data or condition that you specify does not exist; that is, you can code the following clause:

```
WHERE NOT EXISTS (SELECT ...);
```

Related tasks:

 [Writing efficient subqueries \(DB2 Performance\)](#)

Related reference:

 [Quantified predicate \(DB2 SQL\)](#)

 [having-clause \(DB2 SQL\)](#)

 [where-clause \(DB2 SQL\)](#)

 [EXISTS predicate \(DB2 SQL\)](#)

 [IN predicate \(DB2 SQL\)](#)

Correlated subqueries

A *correlated subquery* is a subquery that DB2 reevaluates when it examines a new row (in a WHERE clause) or a group of rows (in a HAVING clause) as it executes the outer SELECT statement.

In an uncorrelated subquery, DB2 executes the subquery once, substitutes the result of the subquery in the right side of the search condition, and evaluates the outer SELECT based on the value of the search condition.

User-defined functions in correlated subqueries

Use care when you invoke a user-defined function in a correlated subquery, and that user-defined function uses a scratchpad. DB2 does not refresh the scratchpad between invocations of the subquery. This can cause undesirable results because the scratchpad keeps values across the invocations of the subquery.

An example of a correlated subquery

Suppose that you want a list of all the employees whose education levels are higher than the average education levels in their respective departments. To get this information, DB2 must search the DSN8A10.EMP table. For each employee in the table, DB2 needs to compare the employee's education level to the average education level for that employee's department.

For this example, you need to use a correlated subquery, which differs from an uncorrelated subquery. An uncorrelated subquery compares the employee's education level to the average of the entire company, which requires looking at the entire table. A correlated subquery evaluates only the department that corresponds to the particular employee.

In the subquery, you tell DB2 to compute the average education level for the department number in the current row. The following query performs this action:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
      FROM DSN8A10.EMP X
     WHERE EDLEVEL >
           (SELECT AVG(EDLEVEL)
            FROM DSN8A10.EMP
            WHERE WORKDEPT = X.WORKDEPT);
```

A correlated subquery looks like an uncorrelated one, except for the presence of one or more correlated references. In the example, the single correlated reference is the occurrence of X.WORKDEPT in the WHERE clause of the subselect. In this clause, the qualifier X is the correlation name that is defined in the FROM clause of the outer SELECT statement. X designates rows of the first instance of DSN8A10.EMP. At any time during the execution of the query, X designates the row of DSN8A10.EMP to which the WHERE clause is being applied.

Consider what happens when the subquery executes for a given row of DSN8A10.EMP. Before it executes, X.WORKDEPT receives the value of the WORKDEPT column for that row. Suppose, for example, that the row is for Christine Haas. Her work department is A00, which is the value of WORKDEPT for that row. Therefore, the following is the subquery that is executed for that row:

```
(SELECT AVG(EDLEVEL)
 FROM DSN8A10.EMP
 WHERE WORKDEPT = 'A00');
```

The subquery produces the average education level of Christine's department. The outer SELECT then compares this average to Christine's own education level. For some other row for which WORKDEPT has a different value, that value appears in the subquery in place of A00. For example, in the row for Michael L Thompson, this value is B01, and the subquery for his row delivers the average education level for department B01.

The result table that is produced by the query is similar to the following output:

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
=====	=====	=====	=====
000010	HASS	A00	18
000030	KWAN	C01	20
000070	PULASKI	D21	16
000090	HENDERSON	E11	16

Related concepts:

[Correlated and non-correlated subqueries \(DB2 Performance\)](#)

Related reference:

[having-clause \(DB2 SQL\)](#)

[where-clause \(DB2 SQL\)](#)

Correlation names in references

A correlation name is a name that you specify for a table, view, nested table expression or table function. This name is valid only within the context in which it is defined. Use correlation names to avoid ambiguity, to establish correlated references, or to use shorter names for tables or views.

A correlated reference can appear in a subquery, in a nested table expression, or as an argument of a user-defined table function. For information about correlated references in nested table expressions and table functions, see “Joining data from more than one table” on page 695. In a subquery, the reference should be of the form *X.C*, where *X* is a correlation name and *C* is the name of a column in the table that *X* represents.

Any number of correlated references can appear in a subquery, with no restrictions on variety. For example, you can use one correlated reference in the outer SELECT, and another in a nested subquery.

When you use a correlated reference in a subquery, the correlation name can be defined in the outer SELECT or in any of the subqueries that contain the reference. Suppose, for example, that a query contains subqueries A, B, and C, and that A contains B and B contains C. The subquery C can use a correlation reference that is defined in B, A, or the outer SELECT.

You can define a correlation name for each table name in a FROM clause. Specify the correlation name after its table name. Leave one or more blanks between a table name and its correlation name. You can include the word AS between the table name and the correlation name to increase the readability of the SQL statement.

The following example demonstrates the use of a correlated reference in the search condition of a subquery:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM DSN8A10.EMP AS X
WHERE EDLEVEL >
      (SELECT AVG(EDLEVEL)
       FROM DSN8A10.EMP
       WHERE WORKDEPT = X.WORKDEPT);
```

The following example demonstrates the use of a correlated reference in the select list of a subquery:

```
UPDATE BP1TBL T1
SET (KEY1, CHAR1, VCHAR1) =
    (SELECT VALUE(T2.KEY1,T1.KEY1), VALUE(T2.CHAR1,T1.CHAR1),
     VALUE(T2.VCHAR1,T1.VCHAR1)
```

```

        FROM BP2TBL T2
        WHERE (T2.KEY1 = T1.KEY1))
WHERE KEY1 IN
  (SELECT KEY1
   FROM BP2TBL T3
   WHERE KEY2 > 0);

```

Using correlated subqueries in an UPDATE statement:

Use correlation names in an UPDATE statement to refer to the rows that you are updating. The subquery for which you specified a correlation name is called a *correlated subquery*.

For example, when all activities of a project must complete before September 2006, your department considers that project to be a priority project. Assume that you have added the PRIORITY column to DSN8A10.PROJ. You can use the following SQL statement to evaluate the projects in the DSN8A10.PROJ table, and write a 1 (a flag to indicate PRIORITY) in the PRIORITY column for each priority project:

```

UPDATE DSN8A10.PROJ X
SET PRIORITY = 1
WHERE DATE('2006-09-01') >
  (SELECT MAX(ACENDATE)
   FROM DSN8A10.PROJACT
   WHERE PROJNO = X.PROJNO);

```

As DB2 examines each row in the DSN8A10.PROJ table, it determines the maximum activity end date (the ACENDATE column) for all activities of the project (from the DSN8A10.PROJACT table). If the end date of each activity that is associated with the project is before September 2006, the current row in the DSN8A10.PROJ table qualifies, and DB2 updates it.

Using correlated subqueries in a DELETE statement:

Use correlation names in a DELETE statement to refer to the rows that you are deleting. The subquery for which you specified a correlation name is called a *correlated subquery*. DB2 evaluates the correlated subquery once for each row in the table that is named in the DELETE statement to decide whether to delete the row.

Using tables with no referential constraints:

Suppose that a department considers a project to be complete when the combined amount of time currently spent on it is less than or equal to half of a person's time. The department then deletes the rows for that project from the DSN8A10.PROJ table. In the examples in this topic, PROJ and PROJACT are independent tables; that is, they are separate tables with no referential constraints defined on them.

```

DELETE FROM DSN8A10.PROJ X
WHERE .5 >
  (SELECT SUM(ACSTAFF)
   FROM DSN8A10.PROJACT
   WHERE PROJNO = X.PROJNO);

```

To process this statement, DB2 determines for each project (represented by a row in the DSN8A10.PROJ table) whether the combined staffing for that project is less than 0.5. If it is, DB2 deletes that row from the DSN8A10.PROJ table.

To continue this example, suppose that DB2 deletes a row in the DSN8A10.PROJ table. You must also delete rows that are related to the deleted project in the DSN8A10.PROJACT table. To do this, use a statement similar to this statement:


```
DELETE FROM DSN8A10.PROJACT X
  WHERE NOT EXISTS
    (SELECT *
     FROM DSN8A10.PROJ
     WHERE PROJNO = X.PROJNO);
```

DB2 determines, for each row in the DSN8A10.PROJACT table, whether a row with the same project number exists in the DSN8A10.PROJ table. If not, DB2 deletes the row from DSN8A10.PROJACT.

Using a single table:

A subquery of a searched DELETE statement (a DELETE statement that does not use a cursor) can reference the same table from which rows are deleted. In the following statement, which deletes the employee with the highest salary from each department, the employee table appears in the outer DELETE and in the subselect:

```
DELETE FROM YEMP X
  WHERE SALARY = (SELECT MAX(SALARY) FROM YEMP Y
                 WHERE X.WORKDEPT =Y.WORKDEPT);
```

This example uses a copy of the employee table for the subquery.

The following statement, without a correlated subquery, yields equivalent results:

```
DELETE FROM YEMP
  WHERE (SALARY, WORKDEPT) IN (SELECT MAX(SALARY), WORKDEPT
                              FROM YEMP
                              GROUP BY WORKDEPT);
```

Using tables with referential constraints:

DB2 restricts delete operations for dependent tables that are involved in referential constraints. If a DELETE statement has a subquery that references a table that is involved in the deletion, make the last delete rule in the path to that table RESTRICT or NO ACTION. This action ensures that the result of the subquery is not materialized before the deletion occurs. However, if the result of the subquery is materialized before the deletion, the delete rule can also be CASCADE or SET NULL.

Example: Without referential constraints, the following statement deletes departments from the department table whose managers are not listed correctly in the employee table:

```
DELETE FROM DSN8A10.DEPT THIS
  WHERE NOT DEPTNO =
    (SELECT WORKDEPT
     FROM DSN8A10.EMP
     WHERE EMPNO = THIS.MGRNO);
```

With the referential constraints that are defined for the sample tables, this statement causes an error because the result table for the subquery is not materialized before the deletion occurs. Because DSN8A10.EMP is a dependent table of DSN8A10.DEPT, the deletion involves the table that is referred to in the subquery, and the last delete rule in the path to EMP is SET NULL, not RESTRICT or NO ACTION. If the statement could execute, its results would depend on the order in which DB2 accesses the rows. Therefore, DB2 prohibits the deletion.

Restrictions when using distinct types with UNION, EXCEPT, and INTERSECT

DB2 enforces strong typing of distinct types with UNION, EXCEPT, and INTERSECT. When you use these keywords to combine column values from several tables, the combined columns must be of the same types. If a column is a distinct type, the corresponding column must be the same distinct type.

Example: Suppose that you create a view that combines the values of the US_SALES, EUROPEAN_SALES, and JAPAN_SALES tables. The TOTAL columns in the three tables are of different distinct types. Before you combine the table values, you must convert the types of two of the TOTAL columns to the type of the third TOTAL column. Assume that the US_DOLLAR type has been chosen as the common distinct type. Because DB2 does not generate cast functions to convert from one distinct type to another, two user-defined functions must exist:

- A function called EURO_TO_US that converts values of type EURO to type US_DOLLAR
- A function called YEN_TO_US that converts values of type JAPANESE_YEN to type US_DOLLAR

Then you can execute a query like this to display a table of combined sales:

```
SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
FROM US_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, EURO_TO_US(TOTAL)
FROM EUROPEAN_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, YEN_TO_US(TOTAL)
FROM JAPAN_SALES;
```

Because the result type of both the YEN_TO_US function and the EURO_TO_US function is US_DOLLAR, you have satisfied the requirement that the distinct types of the combined columns are the same.

Comparison of distinct types

You can compare an object with a distinct type only to an object with exactly the same distinct type. You cannot compare data of a distinct type directly to data of its source type. However, you can compare a distinct type to its source type by using a cast function.

The basic rule for comparisons is that the data types of the operands must be compatible. The compatibility rule defines, for example, that all numeric types (SMALLINT, INTEGER, FLOAT, and DECIMAL) are compatible. That is, you can compare an INTEGER value with a value of type FLOAT. However, you cannot compare an object of a distinct type to an object of a different type. You can compare an object with a distinct type only to an object with exactly the same distinct type.

For example, suppose you want to know which products sold more than \$100 000.00 in the US in the month of July in 2003 (7/03). Because you cannot compare data of type US_DOLLAR with instances of data of the source type of US_DOLLAR (DECIMAL) directly, you must use a cast function to cast data from DECIMAL to US_DOLLAR or from US_DOLLAR to DECIMAL. Whenever you create a distinct type, DB2 creates two cast functions, one to cast from the source type to the distinct type and the other to cast from the distinct type to the source type. For distinct type US_DOLLAR, DB2 creates a cast function called DECIMAL and a cast function called US_DOLLAR. When you compare an object of type

US_DOLLAR to an object of type DECIMAL, you can use one of those cast functions to make the data types identical for the comparison. Suppose table US_SALES is defined like this:

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1990),
   TOTAL         US_DOLLAR);
```

Then you can cast DECIMAL data to US_DOLLAR like this:

```
SELECT PRODUCT_ITEM
FROM   US_SALES
WHERE  TOTAL > US_DOLLAR(100000.00)
AND    MONTH = 7
AND    YEAR  = 2003;
```

The casting satisfies the requirement that the compared data types are identical.

You cannot use host variables in statements that you prepare for dynamic execution. As explained in “Dynamically executing an SQL statement by using PREPARE and EXECUTE” on page 221, you can substitute parameter markers for host variables when you prepare a statement, and then use host variables when you execute the statement.

If you use a parameter marker in a predicate of a query, and the column to which you compare the value represented by the parameter marker is of a distinct type, you must cast the parameter marker to the distinct type, or cast the column to its source type.

For example, suppose that distinct type CNUM is defined like this:

```
CREATE DISTINCT TYPE CNUM AS INTEGER;
```

Table CUSTOMER is defined like this:

```
CREATE TABLE CUSTOMER
  (CUST_NUM  CNUM NOT NULL,
   FIRST_NAME CHAR(30) NOT NULL,
   LAST_NAME  CHAR(30) NOT NULL,
   PHONE_NUM  CHAR(20) WITH DEFAULT,
   PRIMARY KEY (CUST_NUM));
```

In an application program, you prepare a SELECT statement that compares the CUST_NUM column to a parameter marker. Because CUST_NUM is of a distinct type, you must cast the distinct type to its source type:

```
SELECT FIRST_NAME, LAST_NAME, PHONE_NUM FROM CUSTOMER
WHERE CAST(CUST_NUM AS INTEGER) = ?
```

Alternatively, you can cast the parameter marker to the distinct type:

```
SELECT FIRST_NAME, LAST_NAME, PHONE_NUM FROM CUSTOMER
WHERE CUST_NUM = CAST (? AS CNUM)
```

Nested SQL statements

An SQL statement can explicitly invoke user-defined functions or stored procedures or can implicitly activate triggers that invoke user-defined functions or stored procedures. This situation is known as nesting of SQL statements. DB2 supports as many as to 64 levels of nesting.

The following example shows SQL statement nesting.

Trigger TR1 is defined on table T3:

```
CREATE TRIGGER TR1
  AFTER UPDATE ON T3
  FOR EACH STATEMENT MODE DB2SQL
  BEGIN ATOMIC
    CALL SP3(PARM1);
  END
```

Program P1 (nesting level 1) contains:

```
SELECT UDF1(C1) FROM T1;
```

UDF1 (nesting level 2) contains:

```
CALL SP2(C2);
```

SP2 (nesting level 3) contains:

```
UPDATE T3 SET C3=1;
```

SP3 (nesting level 4) contains:

```
SELECT UDF4(C4) FROM T4;
```

```
:
```

SP16 (nesting level 16) cannot invoke stored procedures
or user-defined functions

Be aware of the following DB2 restrictions on nested SQL statements:

- Restrictions for SELECT statements:

When you execute a SELECT statement on a table, you cannot execute INSERT, UPDATE, MERGE, or DELETE statements on the same table at a lower level of nesting.

For example, suppose that you execute this SQL statement at level 1 of nesting:

```
SELECT UDF1(C1) FROM T1;
```

You cannot execute this SQL statement at a lower level of nesting:

```
INSERT INTO T1 VALUES(...);
```

- Restrictions for SELECT FROM FINAL TABLE statements that specify INSERT, UPDATE, or DELETE statements to change data:

When you execute this type of statement, an error occurs if both of the following conditions exist:

- The SELECT statement that modifies data (by specifying INSERT, UPDATE, or DELETE) activates an AFTER TRIGGER.
- The AFTER TRIGGER results in additional nested SQL operations that modify the table that is the target of the original SELECT statement that modifies data.

- Restrictions for INSERT, UPDATE, MERGE, and DELETE statements:

When you execute an INSERT, UPDATE, MERGE, or DELETE statement on a table, you cannot access that table from a user-defined function or stored procedure that is at a lower level of nesting.

For example, suppose that you execute this SQL statement at level 1 of nesting:

```
DELETE FROM T1 WHERE UDF3(T1.C1) = 3;
```

You cannot execute this SELECT statement at a lower level of nesting:

```
SELECT * FROM T1;
```

If the AFTER trigger is not activated by an INSERT, UPDATE, or DELETE data change statement that is specified in a data-change-table-reference SELECT FROM FINAL TABLE, the preceding list of restrictions do not apply to SQL statements that are executed at a lower level of nesting as a result of an after trigger. For example, suppose an UPDATE statement at nesting level 1 activates an after update trigger, which calls a stored procedure. The stored procedure executes two SQL statements that reference the triggering table: one SELECT statement and one INSERT statement. In this situation, both the SELECT and the INSERT statements can be executed even though they are at nesting level 3.

Although trigger activations count in the levels of SQL statement nesting, the previous restrictions on SQL statements do not apply to SQL statements that are executed in the trigger body.

Example: Suppose that trigger TR1 is defined on table T1:

```
CREATE TRIGGER TR1
AFTER INSERT ON T1
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
  UPDATE T1 SET C1=1;
END
```

Now suppose that you execute this SQL statement at level 1 of nesting:

```
INSERT INTO T1 VALUES(...);
```

Although the UPDATE statement in the trigger body is at level 2 of nesting and modifies the same table that the triggering statement updates, DB2 can execute the INSERT statement successfully.

Retrieving a set of rows by using a cursor

In an application program, you can retrieve a set of rows from a table or a result table that is returned by a stored procedure. You can retrieve one or more rows at a time.

Use either of the following types of cursors to retrieve rows from a result table:

- A row-positioned cursor retrieves at most a single row at a time from the result table into host variables. At any point in time, the cursor is positioned on at most a single row. For information about how to use a row-positioned cursor, see “Accessing data by using a row-positioned cursor” on page 726.
- A rowset-positioned cursor retrieves zero, one, or more rows at a time, as a rowset, from the result table into host variable arrays. At any point in time, the cursor can be positioned on a rowset. You can reference all of the rows in the rowset, or only one row in the rowset, when you use a positioned DELETE or positioned UPDATE statement. For information about how to use a rowset-positioned cursor, see “Accessing data by using a rowset-positioned cursor” on page 731.

Cursors

A *cursor* is a mechanism that points to one or more rows in a set of rows. The rows are retrieved from a table or in a result set that is returned by a stored procedure. Your application program can use a cursor to retrieve rows from a table.

Cursors bound with cursor stability that are used in block fetch operations are particularly vulnerable to reading data that has already changed. In a block fetch, database access prefetches rows ahead of the row retrieval controlled by the application. During that time the cursor might close, and the locks might be released, before the application receives the data. Thus, it is possible for the application to fetch a row of values that no longer exists, or to miss a recently inserted row. In many cases, that is acceptable; a case for which it is **not** acceptable is said to require *data currency*.

If your application requires data currency for a cursor, you need to prevent block fetching for the data to which it points. To prevent block fetching for a distributed cursor, declare the cursor with the FOR UPDATE clause.

Types of cursors

You can declare row-positioned or rowset-positioned cursors in a number of ways. These cursors can be scrollable or not scrollable, held or not held, or returnable or not returnable.

In addition, you can declare a returnable cursor in a stored procedure by including the WITH RETURN clause; the cursor can return result sets to a caller of the stored procedure.

Scrollable and non-scrollable cursors:

When you declare a cursor, you tell DB2 whether you want the cursor to be scrollable or non-scrollable by including or omitting the SCROLL clause. This clause determines whether the cursor moves sequentially forward through the result table or can move randomly through the result table.

Using a non-scrollable cursor:

The simplest type of cursor is a non-scrollable cursor. A non-scrollable cursor can be either row-positioned or rowset-positioned. A row-positioned non-scrollable cursor moves forward through its result table one row at a time. Similarly, a rowset-positioned non-scrollable cursor moves forward through its result table one rowset at a time.

A non-scrollable cursor always moves sequentially forward in the result table. When the application opens the cursor, the cursor is positioned before the first row (or first rowset) in the result table. When the application executes the first FETCH, the cursor is positioned on the first row (or first rowset). When the application executes subsequent FETCH statements, the cursor moves one row ahead (or one rowset ahead) for each FETCH. After each FETCH statement, the cursor is positioned on the row (or rowset) that was fetched.

After the application executes a positioned UPDATE or positioned DELETE statement, the cursor stays at the current row (or rowset) of the result table. You cannot retrieve rows (or rowsets) backward or move to a specific position in a result table with a non-scrollable cursor.

Using a scrollable cursor:

To make a cursor scrollable, you declare it as scrollable. A scrollable cursor can be either row-positioned or rowset-positioned. To use a scrollable cursor, you execute FETCH statements that indicate where you want to position the cursor.

If you want to order the rows of the cursor's result set, and you also want the cursor to be updatable, you need to declare the cursor as scrollable, even if you use it only to retrieve rows (or rowsets) sequentially. You can use the ORDER BY clause in the declaration of an updatable cursor only if you declare the cursor as scrollable.

Declaring a scrollable cursor:

To indicate that a cursor is scrollable, you declare it with the SCROLL keyword. The following examples show a characteristic of scrollable cursors: the *sensitivity*.

The following figure shows a declaration for an insensitive scrollable cursor.

```
EXEC SQL DECLARE C1 INSENSITIVE SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8A10.DEPT
  ORDER BY DEPTNO
END-EXEC.
```

Declaring a scrollable cursor with the `INSENSITIVE` keyword has the following effects:

- The size, the order of the rows, and the values for each row of the result table do not change after the application opens the cursor.
- The result table is read-only. Therefore, you cannot declare the cursor with the `FOR UPDATE` clause, and you cannot use the cursor for positioned update or delete operations.

The following figure shows a declaration for a sensitive static scrollable cursor.

```
EXEC SQL DECLARE C2 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8A10.DEPT
  ORDER BY DEPTNO
END-EXEC.
```

Declaring a cursor as `SENSITIVE STATIC` has the following effects:

- When the application executes positioned `UPDATE` and `DELETE` statements with the cursor, those changes are visible in the result table.
- When the current value of a row no longer satisfies the `SELECT` statement that was used in the cursor declaration, that row is no longer visible in the result table.
- When a row of the result table is deleted from the underlying table, that row is no longer visible in the result table.
- Changes that are made to the underlying table by other cursors or other application processes can be visible in the result table, depending on whether the `FETCH` statements that you use with the cursor are `FETCH INSENSITIVE` or `FETCH SENSITIVE` statements.

The following figure shows a declaration for a sensitive dynamic scrollable cursor.

```
EXEC SQL DECLARE C2 SENSITIVE DYNAMIC SCROLL CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DSN8A10.DEPT
  ORDER BY DEPTNO
END-EXEC.
```

Declaring a cursor as `SENSITIVE DYNAMIC` has the following effects:

- When the application executes positioned `UPDATE` and `DELETE` statements with the cursor, those changes are visible. In addition, when the application executes insert, update, or delete operations (within the application but outside the cursor), those changes are visible.
- All committed inserts, updates, and deletes by other application processes are visible.
- Because the `FETCH` statement executes against the base table, the cursor needs no temporary result table. When you define a cursor as `SENSITIVE DYNAMIC`, you cannot specify the `INSENSITIVE` keyword in a `FETCH` statement for that cursor.

- If you specify an ORDER BY clause for a SENSITIVE DYNAMIC cursor, DB2 might choose an index access path if the ORDER BY is fully satisfied by an existing index. However, a dynamic scrollable cursor that is declared with an ORDER BY clause is not updatable.

Static scrollable cursor:

Both the INSENSITIVE cursor and the SENSITIVE STATIC cursor follow the *static cursor model*:

- The size of the result table does not grow after the application opens the cursor. Rows that are inserted into the underlying table are not added to the result table.
- The order of the rows does not change after the application opens the cursor. If the cursor declaration contains an ORDER BY clause, and the columns that are in the ORDER BY clause are updated after the cursor is opened, the order of the rows in the result table does not change.

Dynamic scrollable cursor:

When you declare a cursor as SENSITIVE, you can declare it either STATIC or DYNAMIC. The SENSITIVE DYNAMIC cursor follows the *dynamic cursor model*:

- The size and contents of the result table can change with every fetch. The base table can change while the cursor is scrolling on it. If another application process changes the data, the cursor sees the newly changed data when it is committed. If the application process of the cursor changes the data, the cursor sees the newly changed data immediately.
- The order of the rows can change after the application opens the cursor. If the cursor declaration contains an ORDER BY clause, and columns that are in the ORDER BY clause are updated after the cursor is opened, the order of the rows in the result table changes.

Related concepts:

“FETCH statement interaction between row and rowset positioning” on page 748

Held and non-held cursors

A held cursor does not close after a commit operation. A cursor that is not held closes after a commit operation. You specify whether you want a cursor to be held or not held by including or omitting the WITH HOLD clause when you declare the cursor.

After a commit operation, the position of a held cursor depends on its type:

- A non-scrollable cursor that is held is positioned after the last retrieved row and before the next logical row. The next row can be returned from the result table with a FETCH NEXT statement.
- A static scrollable cursor that is held is positioned on the last retrieved row. The last retrieved row can be returned from the result table with a FETCH CURRENT statement.
- A dynamic scrollable cursor that is held is positioned after the last retrieved row and before the next logical row. The next row can be returned from the result table with a FETCH NEXT statement. DB2 returns SQLCODE +231 for a FETCH CURRENT statement.

A held cursor can close when:

- You issue a CLOSE cursor, ROLLBACK, or CONNECT statement

- You issue a CAF CLOSE function call or an RRSAF TERMINATE THREAD function call
- The application program terminates.

If the program abnormally terminates, the cursor position is lost. To prepare for restart, your program must reposition the cursor.

The following restrictions apply to cursors that are declared WITH HOLD:

- Do not use DECLARE CURSOR WITH HOLD with the new user signon from a DB2 attachment facility, because all open cursors are closed.
- Do not declare a WITH HOLD cursor in a thread that might become inactive. If you do, its locks are held indefinitely.

IMS

You **cannot** use DECLARE CURSOR...WITH HOLD in message processing programs (MPP) and message-driven batch message processing (BMP). Each message is a new user for DB2; whether or not you declare them using WITH HOLD, no cursors continue for new users. You can use WITH HOLD in non-message-driven BMP and DL/I batch programs.

CICS

In CICS applications, you can use DECLARE CURSOR...WITH HOLD to indicate that a cursor should not close at a commit or sync point. However, SYNCPOINT ROLLBACK closes all cursors, and end-of-task (EOT) closes all cursors before DB2 reuses or terminates the thread. Because pseudo-conversational transactions usually have multiple EXEC CICS RETURN statements and thus span multiple EOTs, the scope of a held cursor is limited. Across EOTs, you must reopen and reposition a cursor declared WITH HOLD, as if you had not specified WITH HOLD.

You should always close cursors that you no longer need. If you let DB2 close a CICS attachment cursor, the cursor might not close until the CICS attachment facility reuses or terminates the thread.

If the CICS application is using a protected entry thread, this thread will continue to hold resources, even when the task that has used these resources ends. These resources will not be released until the protected thread terminates.

The following cursor declaration causes the cursor to maintain its position in the DSN8A10.EMP table after a commit point:

```
EXEC SQL
  DECLARE EMPLUPDT CURSOR WITH HOLD FOR
  SELECT EMPNO, LASTNAME, PHONENO, JOB, SALARY, WORKDEPT
  FROM DSN8A10.EMP
  WHERE WORKDEPT < 'D11'
  ORDER BY EMPNO
END-EXEC.
```

Accessing data by using a row-positioned cursor

A row-positioned cursor is a cursor that points to a single row and retrieves at most a single row at a time from the result table. You can specify a fetch request to specify which rows to retrieve, relative to the current cursor position.

To access data by using a row-positioned cursor:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See “Declaring a row cursor.”
2. Execute an OPEN CURSOR to make the cursor available to the application. See “Opening a row cursor” on page 729.
3. Specify what the program is to do when all rows have been retrieved. See “Specifying the action that the row cursor is to take when it reaches the end of the data” on page 729.
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See “Executing SQL statements by using a row cursor” on page 729.
5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. See “Closing a row cursor” on page 731.

Your program can have several cursors, each of which performs the previous steps.

Declaring a row cursor

Before you can use a row-positioned cursor to retrieve rows, you must declare the cursor. When you declare a cursor, you identify a set of rows that are to be accessed with the cursor.

To declare a row cursor, issue a DECLARE CURSOR statement. The DECLARE CURSOR statement names a cursor and specifies a SELECT statement. The SELECT statement defines the criteria for the rows that are to make up the result table.

The following example shows a simple form of the DECLARE CURSOR statement:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
    FROM DSN8A10.EMP
  END-EXEC.
```

You can use this cursor to list select information about employees.

More complicated cursors might include WHERE clauses or joins of several tables. For example, suppose that you want to use a cursor to list employees who work on a certain project. Declare a cursor like this to identify those employees:

```
EXEC SQL
  DECLARE C2 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
    FROM DSN8A10.EMP X
    WHERE EXISTS
      (SELECT *
       FROM DSN8A10.PROJ Y
       WHERE X.EMPNO=Y.REPEMP
       AND Y.PROJNO=:GOODPROJ);
```

Declaring cursors for tables that use multilevel security: You can declare a cursor that retrieves rows from a table that uses multilevel security with row-level granularity. However, the result table for the cursor contains only those rows that have a security label value that is equivalent to or dominated by the security label value of your ID.

Updating a column: You can update columns in the rows that you retrieve. Updating a row after you use a cursor to retrieve it is called a *positioned* update. If you intend to perform any positioned updates on the identified table, include the FOR UPDATE clause. The FOR UPDATE clause has two forms:

- The first form is FOR UPDATE OF *column-list*. Use this form when you know in advance which columns you need to update.
- The second form is FOR UPDATE, with no column list. Use this form when you might use the cursor to update any of the columns of the table.

For example, you can use this cursor to update only the SALARY column of the employee table:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8A10.EMP X
      WHERE EXISTS
        (SELECT *
         FROM DSN8A10.PROJ Y
         WHERE X.EMPNO=Y.RESPEMP
         AND Y.PROJNO=:GOODPROJ)
    FOR UPDATE OF SALARY;
```

If you might use the cursor to update any column of the employee table, define the cursor like this:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8A10.EMP X
      WHERE EXISTS
        (SELECT *
         FROM DSN8A10.PROJ Y
         WHERE X.EMPNO=Y.RESPEMP
         AND Y.PROJNO=:GOODPROJ)
    FOR UPDATE;
```

DB2 must do more processing when you use the FOR UPDATE clause without a column list than when you use the FOR UPDATE clause with a column list. Therefore, if you intend to update only a few columns of a table, your program can run more efficiently if you include a column list.

The precompiler options NOFOR and STDSQL affect the use of the FOR UPDATE clause in static SQL statements. If you do not specify the FOR UPDATE clause in a DECLARE CURSOR statement, and you do not specify the STDSQL(YES) option or the NOFOR precompiler options, you receive an error if you execute a positioned UPDATE statement.

You can update a column of the identified table even though it is not part of the result table. In this case, you do not need to name the column in the SELECT statement. When the cursor retrieves a row (using FETCH) that contains a column value you want to update, you can use UPDATE ... WHERE CURRENT OF to identify the row that is to be updated.

Read-only result table: Some result tables cannot be updated—for example, the result of joining two or more tables.

Related concepts:

 [Multilevel security \(Managing Security\)](#)

Related reference:

“Descriptions of SQL processing options” on page 932

 [DECLARE CURSOR \(DB2 SQL\)](#)

 [select-statement \(DB2 SQL\)](#)

Opening a row cursor

After you declare a row cursor, you need to tell DB2 that you are ready to process the first row of the result table. This action is called opening the cursor.

To open a row cursor, execute the OPEN statement in your program. DB2 then uses the SELECT statement within DECLARE CURSOR to identify a set of rows. If you use host variables in the search condition of that SELECT statement, DB2 uses the **current value** of the variables to select the rows. The result table that satisfies the search condition might contain zero, one, or many rows. An example of an OPEN statement is:

```
EXEC SQL
    OPEN C1
END-EXEC.
```

If you use the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers in a cursor, DB2 determines the values in those special registers only when it opens the cursor. DB2 uses the values that it obtained at OPEN time for all subsequent FETCH statements.

Two factors that influence the amount of time that DB2 requires to process the OPEN statement are:

- Whether DB2 must perform any sorts before it can retrieve rows
- Whether DB2 uses parallelism to process the SELECT statement of the cursor

Specifying the action that the row cursor is to take when it reaches the end of the data

Your program must be coded to recognize and handle an end-of-data condition whenever you use a row cursor to fetch a row.

To determine whether the program has retrieved the last row of data, test the SQLCODE field for a value of 100 or the SQLSTATE field for a value of '02000'. These codes occur when a FETCH statement has retrieved the last row in the result table and your program issues a subsequent FETCH. For example:

```
IF SQLCODE = 100 GO TO DATA-NOT-FOUND.
```

An alternative to this technique is to code the WHENEVER NOT FOUND statement. The WHENEVER NOT FOUND statement causes your program to branch to another part that then issues a CLOSE statement. For example, to branch to label DATA-NOT-FOUND when the FETCH statement does not return a row, use this statement:

```
EXEC SQL
    WHENEVER NOT FOUND GO TO DATA-NOT-FOUND
END-EXEC.
```

For more information about the WHENEVER NOT FOUND statement, see “Checking the execution of SQL statements” on page 227.

Executing SQL statements by using a row cursor

You can use row cursors to execute FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

Execute a FETCH statement for one of the following purposes:

- To copy data from a row of the result table into one or more host variables
- To position the cursor before you perform a positioned update or positioned delete operation

The following example shows a FETCH statement that retrieves selected columns from the employee table:

```
EXEC SQL
  FETCH C1 INTO
    :HV-EMPNO, :HV-FIRSTNME, :HV-MIDINIT, :HV-LASTNAME, :HV-SALARY :IND-SALARY
END-EXEC.
```

The SELECT statement within DECLARE CURSOR statement identifies the result table from which you fetch rows, but DB2 does not retrieve any data until your application program executes a FETCH statement.

When your program executes the FETCH statement, DB2 positions the cursor on a row in the result table. That row is called the *current row*. DB2 then copies the current row contents into the program host variables that you specify on the INTO clause of FETCH. This sequence repeats each time you issue FETCH, until you process all rows in the result table.

The row that DB2 points to when you execute a FETCH statement depends on whether the cursor is declared as a scrollable or non-scrollable.

When you query a remote subsystem with FETCH, consider using block fetch for better performance. Block fetch processes rows ahead of the current row. You cannot use a block fetch when you perform a positioned update or delete operation.

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned UPDATE statement to modify the data in that row. An example of a positioned UPDATE statement is:

```
EXEC SQL
  UPDATE DSN8A10.EMP
    SET SALARY = 50000
    WHERE CURRENT OF C1
END-EXEC.
```

A positioned UPDATE statement updates the row on which the cursor is positioned.

A positioned UPDATE statement is subject to these restrictions:

- You cannot update a row if your update violates any unique, check, or referential constraints.
- You cannot use an UPDATE statement to modify the rows of a created temporary table. However, you can use an UPDATE statement to modify the rows of a declared temporary table.
- If the right side of the SET clause in the UPDATE statement contains a fullselect, that fullselect cannot include a correlated name for a table that is being updated.
- You cannot use an SQL data change statement in the FROM clause of a SELECT statement that defines a cursor that is used in a positioned UPDATE statement.
- A positioned UPDATE statement will fail if the value of the security label column of the row where the cursor is positioned is not equivalent to the security label value of your user id. If your user id has write down privilege, a positioned UPDATE statement will fail if the value of the security label column of the row where the cursor is positioned does not dominate the security label value of your user id.

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned DELETE statement to delete that row. An example of a positioned DELETE statement looks like this:

```
EXEC SQL
  DELETE FROM DSN8A10.EMP
  WHERE CURRENT OF C1
END-EXEC.
```

A positioned DELETE statement deletes the row on which the cursor is positioned.

A positioned DELETE statement is subject to these restrictions:

- You cannot use a DELETE statement with a cursor to delete rows from a created temporary table. However, you can use a DELETE statement with a cursor to delete rows from a declared temporary table.
- After you have deleted a row, you cannot update or delete another row using that cursor until you execute a FETCH statement to position the cursor on another row.
- You cannot delete a row if doing so violates any referential constraints.
- You cannot use an SQL data change statement in the FROM clause of a SELECT statement that defines a cursor that is used in a positioned DELETE statement.
- A positioned DELETE statement will fail if the value of the security label column of the row where the cursor is positioned is not equivalent to the security label value of your user id. If your user id has write down privilege, a positioned DELETE statement will fail if the value of the security label column of the row where the cursor is positioned does not dominate the security label value of your user id.

Closing a row cursor

Close a row cursor when it finishes processing rows if you want to free the resources or if you want to use the cursor again. Otherwise, you can let DB2 automatically close the cursor when the current transaction terminates or when your program terminates.

To free the resources that are held by the cursor, close the cursor explicitly by issuing the CLOSE statement.

If you want to use the rowset cursor again, reopen it.

To close a row cursor:

Issue a CLOSE statement. An example of a CLOSE statement looks like this:

```
EXEC SQL
  CLOSE C1
END-EXEC.
```

Accessing data by using a rowset-positioned cursor

A rowset-positioned cursor is a cursor that can return one or more rows for a single fetch operation. The cursor is positioned on the set of rows that are to be fetched.

To access data by using a rowset-positioned cursor:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See “Declaring a rowset cursor” on page 732.

2. Execute an OPEN CURSOR to make the cursor available to the application. See “Opening a rowset cursor.”
3. Specify what the program is to do when all rows have been retrieved. See “Specifying the action that the rowset cursor is to take when it reaches the end of the data.”
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See “Executing SQL statements by using a rowset cursor” on page 733.
5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. See “Closing a rowset cursor” on page 736.

Your program can have several cursors, each of which performs the previous steps.

Declaring a rowset cursor

Before you can use a rowset-positioned cursor to retrieve rows, you must declare a cursor that is enabled to fetch rowsets. When you declare a cursor, you identify a set of rows that are to be accessed with the cursor.

For restrictions that apply to rowset-positioned cursors and row-positioned cursors, see “Declaring a row cursor” on page 727.

To declare a rowset cursor:

Use the WITH ROWSET POSITIONING clause in the DECLARE CURSOR statement. The following example shows how to declare a rowset cursor:

```
EXEC SQL
  DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR
  SELECT EMPNO, LASTNAME, SALARY
  FROM DSN8A10.EMP
END-EXEC.
```

Opening a rowset cursor

After you declare a rowset cursor, you need to tell DB2 that you are ready to process the first rowset of the result table. This action is called opening the cursor.

To open a rowset cursor, execute the OPEN statement in your program. DB2 then uses the SELECT statement within DECLARE CURSOR to identify the rows in the result table. For more information about the OPEN CURSOR process, see “Opening a row cursor” on page 729.

Specifying the action that the rowset cursor is to take when it reaches the end of the data

Your program must be coded to recognize and handle an end-of-data condition whenever you use a rowset cursor to fetch rows.

To determine whether the program has retrieved the last row of data in the result table, test the SQLCODE field for a value of +100 or the SQLSTATE field for a value of '02000'. With a rowset cursor, these codes occur when a FETCH statement retrieves the last row in the result table. However, when the last row has been retrieved, the program must still process the rows in the last rowset through that last row. For an example of end-of-data processing for a rowset cursor, see “Examples of fetching rows by using cursors” on page 749.

To determine the number of retrieved rows, use either of the following values:

- The contents of the SQLERRD(3) field in the SQLCA
- The contents of the ROW_COUNT item of GET DIAGNOSTICS

For information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234.

If you declare the cursor as dynamic scrollable, and SQLCODE has the value +100, you can continue with a FETCH statement until no more rows are retrieved. Additional fetches might retrieve more rows because a dynamic scrollable cursor is sensitive to updates by other application processes. For information about dynamic cursors, see “Types of cursors” on page 723.

Executing SQL statements by using a rowset cursor

You can use rowset cursors to execute multiple-row FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

You can execute these static SQL statements when you use a rowset cursor:

- A multiple-row FETCH statement that copies a rowset of column values into either of the following data areas:
 - Host variable arrays that are declared in your program
 - Dynamically-allocated arrays whose storage addresses are put into an SQL descriptor area (SQLDA), along with the attributes of the columns that are to be retrieved
- After either form of the multiple-row FETCH statement, you can issue:
 - A positioned UPDATE statement on the current rowset
 - A positioned DELETE statement on the current rowset

You must use the WITH ROWSET POSITIONING clause of the DECLARE CURSOR statement if you plan to use a rowset-positioned FETCH statement.

The following example shows a FETCH statement that retrieves 20 rows into host variable arrays that are declared in your program:

```
EXEC SQL
  FETCH NEXT ROWSET FROM C1
  FOR 20 ROWS
  INTO :HVA-EMPNO, :HVA-LASTNAME, :HVA-SALARY :INDA-SALARY
END-EXEC.
```

When your program executes a FETCH statement with the ROWSET keyword, the cursor is positioned on a rowset in the result table. That rowset is called the *current rowset*. The dimension of each of the host variable arrays must be greater than or equal to the number of rows to be retrieved.

Suppose that you want to dynamically allocate the storage needed for the arrays of column values that are to be retrieved from the employee table. You must:

1. Declare an SQLDA structure and the variables that reference the SQLDA.
2. Dynamically allocate the SQLDA and the arrays needed for the column values.
3. Set the fields in the SQLDA for the column values to be retrieved.
4. Open the cursor.
5. Fetch the rows.

You must first declare the SQLDA structure. The following SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```

Your program must also declare variables that reference the SQLDA structure, the SQLVAR structure within the SQLDA, and the DECLEN structure for the precision and scale if you are retrieving a DECIMAL column. For C programs, the code looks like this:


```

struct sqlda *sqldaptr;
struct sqlvar *varptr;
struct DECLEN {
    unsigned char precision;
    unsigned char scale;
};

```

Before you can set the fields in the SQLDA for the column values to be retrieved, you must dynamically allocate storage for the SQLDA structure. For C programs, the code looks like this:

```
sqldaptr = (struct sqlda *) malloc (3 * 44 + 16);
```

The size of the SQLDA is $SQLN * 44 + 16$, where the value of the SQLN field is the number of output columns.

You must set the fields in the SQLDA structure for your FETCH statement. Suppose you want to retrieve the columns EMPNO, LASTNAME, and SALARY. The C code to set the SQLDA fields for these columns looks like this:

```

strcpy(sqldaptr->sqldaid,"SQLDA");
sqldaptr->sqldabc = 148; /* number bytes of storage allocated for the SQLDA */
sqldaptr->sqln = 3; /* number of SQLVAR occurrences */
sqldaptr->sqld = 3;
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0])); /* Point to first SQLVAR */
varptr->sqltype = 452; /* data type CHAR(6) */
varptr->sqlllen = 6;
varptr->sqldata = (char *) hva1;
varptr->sqlind = (short *) inda1;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\x00\x00\x00\x00\x00\x01\x00\x14",varptr->sqlname.length);
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]) + 1); /* Point to next SQLVAR */
varptr->sqltype = 448; /* data type VARCHAR(15) */
varptr->sqlllen = 15;
varptr->sqldata = (char *) hva2;
varptr->sqlind = (short *) inda2;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\x00\x00\x00\x00\x00\x01\x00\x14",varptr->sqlname.length);
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]) + 2); /* Point to next SQLVAR */
varptr->sqltype = 485; /* data type DECIMAL(9,2) */
((struct DECLEN *) &(varptr->sqlllen))->precision = 9;
((struct DECLEN *) &(varptr->sqlllen))->scale = 2;
varptr->sqldata = (char *) hva3;
varptr->sqlind = (short *) inda3;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\x00\x00\x00\x00\x00\x01\x00\x14",varptr->sqlname.length);

```

The SQLDA structure has these fields:

- SQLDABC indicates the number of bytes of storage that are allocated for the SQLDA. The storage includes a 16-byte header and 44 bytes for each SQLVAR field. The value is $SQLN * 44 + 16$, or 148 for this example.
- SQLN is the number of SQLVAR occurrences (or the number of output columns).
- SQLD is the number of variables in the SQLDA that are used by DB2 when processing the FETCH statement.
- Each SQLVAR occurrence describes a host variable array or buffer into which the values for a column in the result table are to be returned. Within each SQLVAR:
 - SQLTYPE indicates the data type of the column.
 - SQLLEN indicates the length of the column. If the data type is DECIMAL, this field has two parts: the PRECISION and the SCALE.

- SQLDATA points to the first element of the array for the column values. For this example, assume that your program allocates the dynamic variable arrays hva1, hva2, and hva3, and their indicator arrays inda1, inda2, and inda3.
- SQLIND points to the first element of the array of indicator values for the column. If SQLTYPE is an odd number, this attribute is required. (If SQLTYPE is an odd number, null values are allowed for the column.)
- SQLNAME has two parts: the LENGTH and the DATA. The LENGTH is 8. The first two bytes of the DATA field is X'0000'. Bytes 5 and 6 of the DATA field are a flag indicating whether the variable is an array or a FOR n ROWS value. Bytes 7 and 8 are a two-byte binary integer representation of the dimension of the array.

You can open the cursor only after all of the fields have been set in the output SQLDA:

```
EXEC SQL OPEN C1;
```

After the OPEN statement, the program fetches the next rowset:

```
EXEC SQL
  FETCH NEXT ROWSET FROM C1
  FOR 20 ROWS
  USING DESCRIPTOR :*sqldaptr;
```

The USING clause of the FETCH statement names the SQLDA that describes the columns that are to be retrieved.

After your program executes a FETCH statement to establish the current rowset, you can use a positioned UPDATE statement with either of the following clauses:

- Use WHERE CURRENT OF to modify all of the rows in the current rowset
- Use FOR ROW *n* OF ROWSET to modify row *n* in the current rowset

An example of a positioned UPDATE statement that uses the WHERE CURRENT OF clause is:

```
EXEC SQL
  UPDATE DSN8A10.EMP
  SET SALARY = 50000
  WHERE CURRENT OF C1
END-EXEC.
```

When the UPDATE statement is executed, the cursor must be positioned on a row or rowset of the result table. If the cursor is positioned on a row, that row is updated. If the cursor is positioned on a rowset, all of the rows in the rowset are updated.

An example of a positioned UPDATE statement that uses the FOR ROW *n* OF ROWSET clause is:

```
EXEC SQL
  UPDATE DSN8A10.EMP
  SET SALARY = 50000
  FOR CURSOR C1 FOR ROW 5 OF ROWSET
END-EXEC.
```

When the UPDATE statement is executed, the cursor must be positioned on a rowset of the result table. The specified row (in the example, row 5) of the current rowset is updated.

After your program executes a FETCH statement to establish the current rowset, you can use a positioned DELETE statement with either of the following clauses:

- Use WHERE CURRENT OF to delete all of the rows in the current rowset
- Use FOR ROW *n* OF ROWSET to delete row *n* in the current rowset

An example of a positioned DELETE statement that uses the WHERE CURRENT OF clause is:

```
EXEC SQL
  DELETE FROM DSN8A10.EMP
  WHERE CURRENT OF C1
END-EXEC.
```

When the DELETE statement is executed, the cursor must be positioned on a row or rowset of the result table. If the cursor is positioned on a row, that row is deleted, and the cursor is positioned before the next row of its result table. If the cursor is positioned on a rowset, all of the rows in the rowset are deleted, and the cursor is positioned before the next rowset of its result table.

An example of a positioned DELETE statement that uses the FOR ROW *n* OF ROWSET clause is:

```
EXEC SQL
  DELETE FROM DSN8A10.EMP
  FOR CURSOR C1 FOR ROW 5 OF ROWSET
END-EXEC.
```

When the DELETE statement is executed, the cursor must be positioned on a rowset of the result table. The specified row of the current rowset is deleted, and the cursor remains positioned on that rowset. The deleted row (in the example, row 5 of the rowset) cannot be retrieved or updated.

Related tasks:

“Including dynamic SQL in your program” on page 193

“Executing SQL statements by using a row cursor” on page 729

Related reference:

 SQL descriptor area (SQLDA) (DB2 SQL)

Specifying the number of rows in a rowset:

If you do not explicitly specify the number of rows in a rowset, DB2 implicitly determines the number of rows based on the last fetch request.

To explicitly set the size of a rowset, use the FOR *n* ROWS clause in the FETCH statement. If a FETCH statement specifies the ROWSET keyword, and not the FOR *n* ROWS clause, the size of the rowset is implicitly set to the size of the rowset that was most recently specified in a prior FETCH statement. If a prior FETCH statement did not specify the FOR *n* ROWS clause or the ROWSET keyword, the size of the current rowset is implicitly set to 1. For examples of rowset positioning, see Table 122 on page 749.

Closing a rowset cursor

Close a rowset cursor when it finishes processing rows if you want to free the resources or if you want to use the cursor again. Otherwise, you can let DB2 automatically close the cursor when the current transaction terminates or when your program terminates.

To free the resources held by the cursor, close the cursor explicitly by issuing the CLOSE statement.

If you want to use the rowset cursor again, reopen it.

To close a rowset cursor:

Issue a CLOSE statement.

Retrieving rows by using a scrollable cursor

A *scrollable cursor* is cursor that can be moved in both a forward and a backward direction. Scrollable cursors can be either row-positioned or rowset-positioned.

When you open any cursor, the cursor is positioned before the first row of the result table. You move a scrollable cursor around in the result table by specifying a *fetch orientation* keyword in a FETCH statement. A fetch orientation keyword indicates the absolute or relative position of the cursor when the FETCH statement is executed. The following table lists the fetch orientation keywords that you can specify and their meanings. These keywords apply to both row-positioned scrollable cursors and rowset-positioned scrollable cursors.

Table 120. Positions for a scrollable cursor

Keyword in FETCH statement	Cursor position when FETCH is executed ¹
BEFORE	Before the first row
FIRST or ABSOLUTE +1	On the first row
LAST or ABSOLUTE -1	On the last row
AFTER	After the last row
ABSOLUTE ²	On an absolute row number, from before the first row forward or from after the last row backward
RELATIVE ²	On the row that is forward or backward a relative number of rows from the current row
CURRENT	On the current row
PRIOR or RELATIVE -1	On the previous row
NEXT	On the next row (default)

Notes:

1. The cursor position applies to both row position and rowset position, for example, before the first row or before the first rowset.
2. For more information about ABSOLUTE and RELATIVE, see the FETCH statement syntax.

Example: To use the cursor that is declared in “Types of cursors” on page 723 to fetch the fifth row of the result table, use a FETCH statement like this:

```
EXEC SQL FETCH ABSOLUTE +5 C1 INTO :HVDEPTNO, :DEPTNAME, :MGRNO;
```

To fetch the fifth row from the end of the result table, use this FETCH statement:

```
EXEC SQL FETCH ABSOLUTE -5 C1 INTO :HVDEPTNO, :DEPTNAME, :MGRNO;
```

Related concepts:

“Types of cursors” on page 723

Related reference:

 FETCH (DB2 SQL)

Comparison of scrollable cursors

Whether a scrollable cursor can view the changes that are made to the data by other processes or cursors depends on how the cursor is declared. It also depends on the type of fetch operation that is executed.

When you declare a cursor as `SENSITIVE STATIC`, changes that other processes or cursors make to the underlying table **can** be visible to the result table of the cursor. Whether those changes **are** visible depends on whether you specify `SENSITIVE` or `INSENSITIVE` when you execute `FETCH` statements with the cursor. When you specify `FETCH INSENSITIVE`, changes that other processes or other cursors make to the underlying table are not visible in the result table. When you specify `FETCH SENSITIVE`, changes that other processes or cursors make to the underlying table are visible in the result table.

When you declare a cursor as `SENSITIVE DYNAMIC`, changes that other processes or cursors make to the underlying table are visible to the result table after the changes are committed.

The following table summarizes the sensitivity values and their effects on the result table of a scrollable cursor.

Table 121. How sensitivity affects the result table for a scrollable cursor

DECLARE sensitivity	FETCH INSENSITIVE	FETCH SENSITIVE
INSENSITIVE	No changes to the underlying table are visible in the result table. Positioned <code>UPDATE</code> and <code>DELETE</code> statements using the cursor are not allowed.	Not valid.
SENSITIVE STATIC	Only positioned updates and deletes that are made by the cursor are visible in the result table.	All updates and deletes are visible in the result table. Inserts made by other processes are not visible in the result table.
SENSITIVE DYNAMIC	Not valid.	All committed changes are visible in the result table, including updates, deletes, inserts, and changes in the order of the rows.

Scrolling through a table in any direction

Use a scrollable cursor to move through the table in both a forward and a backward direction.

Question: How can I fetch rows from a table in any direction?

Answer: Declare your cursor as scrollable. When you select rows from the table, you can use the various forms of the `FETCH` statement to move to an absolute row number, move ahead or back a certain number of rows, to the first or last row, before the first row or after the last row, forward, or backward. You can use any combination of these `FETCH` statements to change direction repeatedly.

You can use code like the following example to move forward in the department table by 10 records, backward five records, and forward again by three records:

```
/*  
*****/  
/* Declare host variables */  
*****/  
*/
```

```

EXEC SQL BEGIN DECLARE SECTION;
    char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/*****
/* Declare scrollable cursor to retrieve department names */
*****/
EXEC SQL DECLARE C1 SCROLL CURSOR FOR
    SELECT DEPTNAME FROM DSN8A10.DEPT;
:
:
/*****
/* Open the cursor and position it before the start of
/* the result table.
*****/
EXEC SQL OPEN C1;
EXEC SQL FETCH BEFORE FROM C1;
/*****
/* Fetch first 10 rows
*****/
for(i=0;i<10;i++)
{
    EXEC SQL FETCH NEXT FROM C1 INTO :hv_deptname;
}
/*****
/* Save the value in the tenth row
*****/
tenth_row=hv_deptname;
/*****
/* Fetch backward 5 rows
*****/
for(i=0;i<5;i++)
{
    EXEC SQL FETCH PRIOR FROM C1 INTO :hv_deptname;
}
/*****
/* Save the value in the fifth row
*****/
fifth_row=hv_deptname;
/*****
/* Fetch forward 3 rows
*****/
for(i=0;i<3;i++)
{
    EXEC SQL FETCH NEXT FROM C1 INTO :hv_deptname;
}
/*****
/* Save the value in the eighth row
*****/
eighth_row=hv_deptname;
/*****
/* Close the cursor
*****/
EXEC SQL CLOSE C1;

```

Determining the number of rows in the result table for a static scrollable cursor

You can determine how many rows are in the result table of an INSENSITIVE or SENSITIVE STATIC scrollable cursor.

To determine the number of rows in the result table for a static scrollable cursor, execute a FETCH statement, such as FETCH AFTER, that positions the cursor after the last row. You can then examine the fields SQLERRD(1) and SQLERRD(2) in the SQLCA (fields sqlerrd[0] and sqlerrd[1] for C and C++) for the number of rows in the result table. Alternatively, you can use the GET DIAGNOSTICS statement to retrieve the number of rows in the ROW_COUNT statement item.

Removing a delete hole or update hole

If you try to fetch data from a delete hole or an update hole, DB2 issues an SQL warning. If you try to update or to delete a delete hole or delete an update hole, DB2 issues an SQL error.

You can remove a delete hole only by opening the scrollable cursor, setting a savepoint, executing a positioned DELETE statement with the scrollable cursor, and rolling back to the savepoint.

You can convert an update hole back to a result table row by updating the row in the base table, as shown in the following figure. You can update the base table with a searched UPDATE statement in the same application process, or a searched or positioned UPDATE statement in another application process. After you update the base table, if the row qualifies for the result table, the update hole disappears.

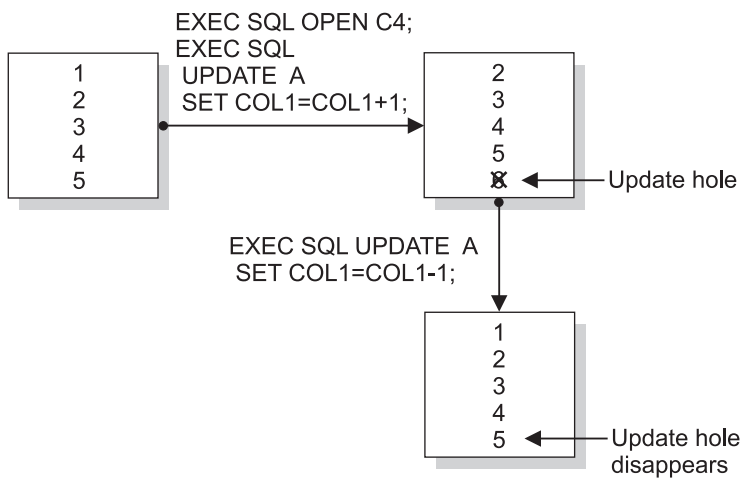


Figure 38. Removing an update hole

A hole becomes visible to a cursor when a cursor operation returns a non-zero SQLCODE. The point at which a hole becomes visible depends on the following factors:

- Whether the scrollable cursor creates the hole
- Whether the FETCH statement is FETCH SENSITIVE or FETCH INSENSITIVE

If the scrollable cursor creates the hole, the hole is visible when you execute a FETCH statement for the row that contains the hole. The FETCH statement can be FETCH INSENSITIVE or FETCH SENSITIVE.

If an update or delete operation outside the scrollable cursor creates the hole, the hole is visible at the following times:

- If you execute a FETCH SENSITIVE statement for the row that contains the hole, the hole is visible when you execute the FETCH statement.
- If you execute a FETCH INSENSITIVE statement, the hole is not visible when you execute the FETCH statement. DB2 returns the row as it was before the update or delete operation occurred. However, if you follow the FETCH INSENSITIVE statement with a positioned UPDATE or DELETE statement, the hole becomes visible.

Holes in the result table of a scrollable cursor:

A hole in the result table means that the result table does not shrink to fill the space of deleted rows. It also does not shrink to fill the space of rows that have been updated and no longer satisfy the search condition. You cannot access a delete or update hole. However, you can remove holes in specific situations.

In some situations, you might not be able to fetch a row from the result table of a scrollable cursor, depending on how the cursor is declared:

- Scrollable cursors that are declared as *INSENSITIVE* or *SENSITIVE STATIC* follow a *static model*, which means that DB2 determines the size of the result table and the order of the rows when you open the cursor.

Deleting or updating rows after a static cursor is open can result in holes in the result table. See “Removing a delete hole or update hole” on page 740.

- Scrollable cursors that are declared as *SENSITIVE DYNAMIC* follow a *dynamic model*, which means that the size and contents of the result table, and the order of the rows, can change after you open the cursor.

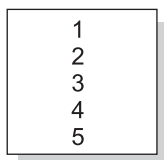
A dynamic cursor scrolls directly on the base table. If the current row of the cursor is deleted or if it is updated so that it no longer satisfies the search condition, and the next cursor operation is *FETCH CURRENT*, then DB2 issues an SQL warning.

The following examples demonstrate how delete and update holes can occur when you use a *SENSITIVE STATIC* scrollable cursor.

Creating a delete hole with a static scrollable cursor:

Suppose that table A consists of one integer column, *COL1*, which has the values shown in the following figure.

Now suppose that you declare the following *SENSITIVE STATIC* scrollable cursor,



1
2
3
4
5

Figure 39. Values for *COL1* of table A

which you use to delete rows from A:

```
EXEC SQL DECLARE C3 SENSITIVE STATIC SCROLL CURSOR FOR
  SELECT COL1
  FROM A
  FOR UPDATE OF COL1;
```

Now you execute the following SQL statements:

```
EXEC SQL OPEN C3;
EXEC SQL FETCH ABSOLUTE +3 C3 INTO :HVCOL1;
EXEC SQL DELETE FROM A WHERE CURRENT OF C3;
```

The positioned delete statement creates a delete hole, as shown in the following figure.

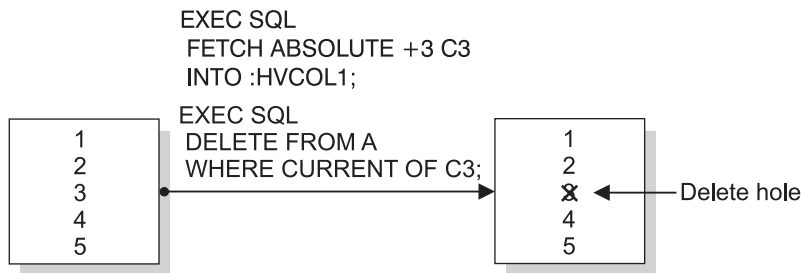


Figure 40. Delete hole

After you execute the positioned delete statement, the third row is deleted from the result table, but the result table does not shrink to fill the space that the deleted row creates.

Creating an update hole with a static scrollable cursor

Suppose that you declare the following SENSITIVE STATIC scrollable cursor, which you use to update rows in A:

```

EXEC SQL DECLARE C4 SENSITIVE STATIC SCROLL CURSOR FOR
SELECT COL1
FROM A
WHERE COL1<6;

```

Now you execute the following SQL statements:

```

EXEC SQL OPEN C4;
UPDATE A SET COL1=COL1+1;

```

The searched UPDATE statement creates an update hole, as shown in the following figure.

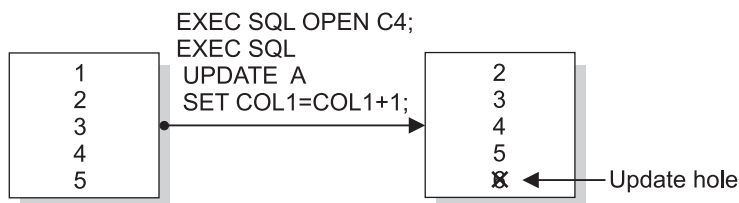


Figure 41. Update hole

After you execute the searched UPDATE statement, the last row no longer qualifies for the result table, but the result table does not shrink to fill the space that the disqualified row creates.

Accessing XML or LOB data quickly by using FETCH WITH CONTINUE

Use the FETCH WITH CONTINUE statement to improve the performance of some queries that reference XML and LOB columns with unknown or very large maximum lengths.

FETCH WITH CONTINUE breaks XML and LOB values into manageable pieces and processes the pieces one at a time to avoid the following buffer allocation problems:

- Allocating overly large or unnecessary space for buffers. If some LOB values are shorter than the maximum length for values in a column, you can waste buffer space if you allocate enough space for the maximum length. The buffer allocation problem can be even worse for XML data because an XML column does not have a defined maximum length. If you use `FETCH WITH CONTINUE`, you can allocate more appropriate buffer space for the actual length of the XML and LOB values.
- Truncating very large XML and LOB data. If a very large XML or LOB value does not fit in the host variable buffer space that is provided by the application program, DB2 truncates the value. If the application program retries this fetch with a larger buffer, two problems exist. First, when using a non-scrollable cursor, you cannot re-fetch the current row without closing, reopening, and repositioning the cursor to the row that was truncated. Second, if you do not use `FETCH WITH CONTINUE`, DB2 does not return the actual length of the entire value to the application program. Thus, DB2 does not know how large a buffer to reallocate. If you use `FETCH WITH CONTINUE`, DB2 preserves the truncated portion of the data for subsequent retrieval and returns the actual length of the entire data value so that the application can reallocate a buffer of the appropriate size.

DB2 provides two methods for using `FETCH WITH CONTINUE` with LOB and XML data:

- “Dynamically allocating buffers when fetching XML and LOB data”
- “Moving data through fixed-size buffers when fetching XML and LOB data” on page 744

Dynamically allocating buffers when fetching XML and LOB data

If you specify `FETCH WITH CONTINUE`, DB2 returns information about which data does not fit in the buffer. Your application can then use the information about the truncated data to allocate an appropriate target buffer and execute a fetch operation with the `CURRENT CONTINUE` clause to retrieve the remaining data.

To use dynamic buffer allocation for LOB and XML data:

1. Use an initial `FETCH WITH CONTINUE` to fetch data into a pre-allocated buffer of a moderate size.
2. If the value is too large to fit in the buffer, use the length information that is returned by DB2 to allocate the appropriate amount of storage.
3. Use a single `FETCH CURRENT CONTINUE` statement to retrieve the remainder of the data.

Suppose that table T1 was created with the following statement:

```
CREATE TABLE T1 (C1 INT, C2 CLOB(100M), C3 CLOB(32K), C4 XML);
```

A row exists in T1 where C1 contains a valid integer, C2 contains 10MB of data, C3 contains 32KB of data, and C4 contains 4MB of data.

Now, suppose that you declare `CURSOR1`, prepare and describe statement `DYNSQLSTMT1` with descriptor `sqlda`, and open `CURSOR1` with the following statements:

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR DYNSQLSTMT1;
EXEC SQL PREPARE DYNSQLSTMT1 FROM 'SELECT * FROM T1';
EXEC SQL DESCRIBE DYNSQLSTMT1 INTO DESCRIPTOR :SQLDA;
EXEC SQL OPEN CURSOR1;
```

Next, suppose that you allocate moderately sized buffers (32 KB for each CLOB or XML column) and set data pointers and lengths in SQLDA. Then, you use the following FETCH WITH CONTINUE statement:

```
EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;
```

Because C2 and C4 contain data that do not fit in the buffer, some of the data is truncated. Your application can use the information that DB2 returns to allocate large enough buffers for the remaining data and reset the data pointers and length fields in SQLDA. At that point, you can resume the fetch and complete the process with the following FETCH CURRENT CONTINUE statement and CLOSE CURSOR statement:

```
EXEC SQL FETCH CURRENT CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;  
EXEC SQL CLOSE CURSOR1;
```

The application needs to concatenate the two returned pieces of the data value. One technique is to move the first piece of data to the dynamically-allocated larger buffer before the FETCH CONTINUE. Set the SQLDATA pointer in the SQLDA structure to point immediately after the last byte of this truncated value. DB2 then writes the remaining data to this location and thus completes the concatenation.

Moving data through fixed-size buffers when fetching XML and LOB data

If you use the WITH CONTINUE clause, DB2 returns information about which data does not fit in the buffer. Your application can then use repeated FETCH CURRENT CONTINUE operations to effectively “stream” large XML and LOB data through a fixed-size buffer, one piece at a time.

To use fixed buffer allocation for LOB and XML data, perform the following steps:

1. Use an initial FETCH WITH CONTINUE to fetch data into a pre-allocated buffer of a moderate size.
2. If the value is too large to fit in the buffer, use as many FETCH CONTINUE statements as necessary to process all of the data through a fixed buffer.

After each FETCH operation, check whether a column was truncated by first examining the SQLWARN1 field in the returned SQLCA. If that field contains a 'W' value, at least one column in the returned row has been truncated. To then determine if a particular LOB or XML column was truncated, your application must compare the value that is returned in the length field with the declared length of the host variable. If a column is truncated, continue to use FETCH CONTINUE statements until all of the data has been retrieved.

After you fetch each piece of the data, move it out of the buffer to make way for the next fetch. Your application can write the pieces to an output file or reconstruct the entire data value in a buffer above the 2-GB bar.

Example: Suppose that table T1 was created with the following statement:

```
CREATE TABLE T1 (C1 INT, C2 CLOB(100M), C3 CLOB(32K), C4 XML);
```

A row exists in T1 where C2 contains 10 MB of data.

Now, suppose that you declare a 32 KB section CLOBHV:

```
EXEC SQL BEGIN DECLARE SECTION  
  DECLARE CLOBHV SQL TYPE IS CLOB(32767);  
EXEC SQL END DECLARE SECTION.
```

Next, suppose that you use the following statements to declare and open CURSOR1 and to FETCH WITH CONTINUE:

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR SELECT C2 FROM T1;
EXEC SQL OPEN CURSOR1;
EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO :CLOBHV;
```

As each piece of the data value is fetched, move it from the buffer to the output file.

Because the 10 MB value in C2 does not fit into the 32 KB buffer, some of the data is truncated. Your application can loop through the following `FETCH CURRENT CONTINUE`:

```
EXEC SQL FETCH CURRENT CONTINUE CURSOR1 INTO :CLOBHV;
```

After each `FETCH` operation, you can determine if the data was truncated by first checking if the `SQLWARN1` field in the returned `SQLCA` contains a 'W' value. If so, then check if the length value, which is returned in `CLOBHV_LENGTH`, is greater than the declared length of 32767. (`CLOBHV_LENGTH` is declared as part of the precompiler expansion of the `CLOBHV` declaration.) If the value is greater, that value has been truncated and more data can be retrieved with the next `FETCH CONTINUE` operation.

When all of the data has moved to the output file, you can close the cursor:

```
EXEC SQL CLOSE CURSOR1;
```

Determining the attributes of a cursor by using the SQLCA

An *SQL communications area (SQLCA)* is an area that is set apart for communication with DB2 and consists of a collection of variables. Using the `SQLCA` is one way to get information about any open cursors. Alternatively, you can use the `GET DIAGNOSTICS` statement.

After you open a cursor, you can determine the following attributes of the cursor by checking the following `SQLWARN` and `SQLERRD` fields of the `SQLCA`:

SQLWARN1

Indicates whether the cursor is scrollable or non-scrollable.

SQLWARN4

Indicates whether the cursor is insensitive (I), sensitive static (S), or sensitive dynamic (D).

SQLWARN5

Indicates whether the cursor is read-only, readable and deletable, or readable, deletable, and updatable.

SQLERRD(1) and SQLERRD(2)

These two fields together contain a double byte integer that represents the number of rows in the result table of a cursor when the cursor is positioned after the last row. The cursor is positioned after the last row when the `SQLCODE` is 100. These fields are not set for dynamic scrollable cursors.

SQLERRD(3)

The number of rows in the result table when the `SELECT` statement of the cursor contains a data change statement.

If the `OPEN` statement executes with no errors or warnings, DB2 does not set `SQLWARN0` when it sets `SQLWARN1`, `SQLWARN4`, or `SQLWARN5`.

Related reference:

 [Description of SQLCA fields \(DB2 SQL\)](#)

Determining the attributes of a cursor by using the GET DIAGNOSTICS statement

Using the GET DIAGNOSTICS statement is one way to get information about any open cursors. Alternatively, you can use the SQLCA.

After you open a cursor, you can determine the following attributes of the cursor by checking these GET DIAGNOSTICS items:

DB2_SQL_ATTR_CURSOR_HOLD

Indicates whether the cursor can be held open across commits (Y or N)

DB2_SQL_ATTR_CURSOR_ROWSET

Indicates whether the cursor can use rowset positioning (Y or N)

DB2_SQL_ATTR_CURSOR_SCROLLABLE

Indicates whether the cursor is scrollable (Y or N)

DB2_SQL_ATTR_CURSOR_SENSITIVITY

Indicates whether the cursor is insensitive or sensitive to changes that are made by other processes (I or S)

DB2_SQL_ATTR_CURSOR_TYPE

Indicates whether the cursor is forward (F) declared static (S for INSENSITIVE or SENSITIVE STATIC) or dynamic (D for SENSITIVE DYNAMIC)

For more information about the GET DIAGNOSTICS statement, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 234.

Scrolling through previously retrieved data

To scroll backward through data, use a scrollable cursor, or use a ROWID column or identity column to retrieve data in reverse order.

Question: When a program retrieves data from the database, how can the program scroll backward through the data?

Answer: Use one of the following techniques:

- Use a scrollable cursor.
- If the table contains a ROWID or an identity column, retrieve the values from that column into an array. Then use the ROWID or identity column values to retrieve the rows in reverse order.

Using a scrollable cursor: Using a scrollable cursor to fetch backward through data involves these basic steps:

1. Declare the cursor with the SCROLL keyword.
2. Open the cursor.
3. Execute a FETCH statement to position the cursor at the end of the result table.
4. In a loop, execute FETCH statements that move the cursor backward and then retrieve the data.
5. When you have retrieved all the data, close the cursor.

You can use code like the following example to retrieve department names in reverse order from table DSN8A10.DEPT:

```
/*  
*****/  
/* Declare host variables */  
*****/  
*/
```

```

EXEC SQL BEGIN DECLARE SECTION;
    char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/*****
/* Declare scrollable cursor to retrieve department names */
*****/
EXEC SQL DECLARE C1 SCROLL CURSOR FOR
    SELECT DEPTNAME FROM DSN8A10.DEPT;
:
:
/*****
/* Open the cursor and position it after the end of the */
/* result table. */
*****/
EXEC SQL OPEN C1;
EXEC SQL FETCH AFTER FROM C1;
/*****
/* Fetch rows backward until all rows are fetched. */
*****/
while(SQLCODE==0) {
    EXEC SQL FETCH PRIOR FROM C1 INTO :hv_deptname;
:
:
}
EXEC SQL CLOSE C1;

```

Using a ROWID or identity column: If your table contains a ROWID column or an identity column, you can use that column to rapidly retrieve the rows in reverse order. When you perform the original SELECT, you can store the ROWID or identity column value for each row you retrieve. Then, to retrieve the values in reverse order, you can execute SELECT statements with a WHERE clause that compares the ROWID or identity column value to each stored value.

For example, suppose you add ROWID column DEPTROWID to table DSN8A10.DEPT. You can use code like the following example to select all department names, then retrieve the names in reverse order:

```

/*****
/* Declare host variables */
*****/
EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS ROWID hv_dept_rowid;
    char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/*****
/* Declare other variables */
*****/
struct rowid_struct {
    short int length;
    char data[40]; /* ROWID variable structure */
}
struct rowid_struct rowid_array[200];
/* Array to hold retrieved */
/* ROWIDs. Assume no more */
/* than 200 rows will be */
/* retrieved. */

short int i,j,n;
/*****
/* Declare cursor to retrieve department names */
*****/
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT DEPTNAME, DEPTROWID FROM DSN8A10.DEPT;
:
:
/*****
/* Retrieve the department name and ROWID from DEPT table */
/* and store the ROWID in an array. */
*****/

```

```

/*****
EXEC SQL OPEN C1;
i=0;
while(SQLCODE==0) {
    EXEC SQL FETCH C1 INTO :hv_deptname, :hv_dept_rowid;
    rowid_array[i].length=hv_dept_rowid.length;
    for(j=0;j<hv_dept_rowid.length;j++)
        rowid_array[i].data[j]=hv_dept_rowid.data[j];
    i++;
}
EXEC SQL CLOSE C1;
n=i-1;          /* Get the number of array elements */
/*****
/* Use the ROWID values to retrieve the department names */
/* in reverse order. */
/*****
for(i=n;i>=0;i--) {
    hv_dept_rowid.length=rowid_array[i].length;
    for(j=0;j<hv_dept_rowid.length;j++)
        hv_dept_rowid.data[j]=rowid_array[i].data[j];
    EXEC SQL SELECT DEPTNAME INTO :hv_deptname
        FROM DSN8A10.DEPT
        WHERE DEPTRROWID=:hv_dept_rowid;
}

```

Updating previously retrieved data

To scroll backward through data and update it, use a scrollable cursor that is declared with the FOR UPDATE clause.

Question: How can you scroll backward and update data that was retrieved previously?

Answer: Use a scrollable cursor that is declared with the FOR UPDATE clause.

To update previously retrieved data:

1. Declare the cursor with the SENSITIVE STATIC SCROLL keywords.
2. Open the cursor.
3. Execute a FETCH statement to position the cursor at the end of the result table.
4. FETCH statements that move the cursor backward, until you reach the row that you want to update.
5. Execute the UPDATE WHERE CURRENT OF statement to update the current row.
6. Repeat steps 4 and 5 until you have update all the rows that you need to.
7. When you have retrieved and updated all the data, close the cursor.

FETCH statement interaction between row and rowset positioning

When you declare a cursor with the WITH ROWSET POSITIONING clause, you can intermix row-positioned FETCH statements with rowset-positioned FETCH statements.

The following table shows the interaction between row and rowset positioning for a scrollable cursor. Assume that you declare the scrollable cursor on a table with 15 rows.

Table 122. Interaction between row and rowset positioning for a scrollable cursor

Keywords in FETCH statement	Cursor position when FETCH is executed
FIRST	On row 1
FIRST ROWSET	On a rowset of size 1, consisting of row 1
FIRST ROWSET FOR 5 ROWS	On a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5
CURRENT ROWSET	On a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5
CURRENT	On row 1
NEXT (default)	On row 2
NEXT ROWSET	On a rowset of size 1, consisting of row 3
NEXT ROWSET FOR 3 ROWS	On a rowset of size 3, consisting of rows 4, 5, and 6
NEXT ROWSET	On a rowset of size 3, consisting of rows 7, 8, and 9
LAST	On row 15
LAST ROWSET FOR 2 ROWS	On a rowset of size 2, consisting of rows 14 and 15
PRIOR ROWSET	On a rowset of size 2, consisting of rows 12 and 13
ABSOLUTE 2	On row 2
ROWSET STARTING AT ABSOLUTE 2 FOR 3 ROWS	On a rowset of size 3, consisting of rows 2, 3, and 4
RELATIVE 2	On row 4
ROWSET STARTING AT ABSOLUTE 2 FOR 4 ROWS	On a rowset of size 4, consisting of rows 2, 3, 4, and 5
RELATIVE -1	On row 1
ROWSET STARTING AT ABSOLUTE 3 FOR 2 ROWS	On a rowset of size 2, consisting of rows 3 and 4
ROWSET STARTING AT RELATIVE 4	On a rowset of size 2, consisting of rows 7 and 8
PRIOR	On row 6
ROWSET STARTING AT ABSOLUTE 13 FOR 5 ROWS	On a rowset of size 3, consisting of rows 13, 14, and 15
FIRST ROWSET	On a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5

Related reference:

 [FETCH \(DB2 SQL\)](#)

Examples of fetching rows by using cursors

You can use SQL statements that you include in a COBOL program to define and use non-scrollable cursor for row-positioned updates, scrollable cursors to retrieve rows backward, non-scrollable cursors for rowset-positioned updates, and scrollable cursors for rowset-positioned operations.

The following example shows how to update a row by using a cursor.

```

*****
* Declare a cursor that will be used to update *
* the JOB column of the EMP table. *
*****
EXEC SQL
  DECLARE THISEMP CURSOR FOR
    SELECT EMPNO, LASTNAME,
           WORKDEPT, JOB
    FROM DSN8A10.EMP
    WHERE WORKDEPT = 'D11'
  FOR UPDATE OF JOB
END-EXEC.
*****
* Open the cursor *
*****
EXEC SQL
  OPEN THISEMP
END-EXEC.
*****
* Indicate what action to take when all rows *
* in the result table have been fetched. *
*****
EXEC SQL
  WHENEVER NOT FOUND
  GO TO CLOSE-THISEMP
END-EXEC.
*****
* Fetch a row to position the cursor. *
*****
EXEC SQL
  FETCH FROM THISEMP
  INTO :EMP-NUM, :NAME2,
       :DEPT, :JOB-NAME
END-EXEC.
*****
* Update the row where the cursor is positioned. *
*****
EXEC SQL
  UPDATE DSN8A10.EMP
  SET JOB = :NEW-JOB
  WHERE CURRENT OF THISEMP
END-EXEC.
:
:
*****
* Branch back to fetch and process the next row. *
*****
:
:
*****
* Close the cursor *
*****
CLOSE-THISEMP.
EXEC SQL
  CLOSE THISEMP
END-EXEC.

```

The following example shows how to retrieve data backward with a cursor.

```

*****
* Declare a cursor to retrieve the data backward *
* from the EMP table. The cursor has access to *
* changes by other processes. *
*****
EXEC SQL
  DECLARE THISEMP SENSITIVE STATIC SCROLL CURSOR FOR
    SELECT EMPNO, LASTNAME, WORKDEPT, JOB
    FROM DSN8A10.EMP
END-EXEC.

```



```

*****
* Open the cursor *
*****
EXEC SQL
  OPEN THISEMP
END-EXEC.
*****
* Indicate what action to take when all rows *
* in the result table have been fetched. *
*****
EXEC SQL
  WHENEVER NOT FOUND GO TO CLOSE-THISEMP
END-EXEC.
*****
* Position the cursor after the last row of the *
* result table. This FETCH statement cannot *
* include the SENSITIVE or INSENSITIVE keyword *
* and cannot contain an INTO clause. *
*****
EXEC SQL
  FETCH AFTER FROM THISEMP
END-EXEC.
*****
* Fetch the previous row in the table. *
*****
EXEC SQL
  FETCH SENSITIVE PRIOR FROM THISEMP
  INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
END-EXEC.
*****
* Check that the fetched row is not a hole *
* (SQLCODE +222). If not, print the contents. *
*****
IF SQLCODE IS GREATER THAN OR EQUAL TO 0 AND
  SQLCODE IS NOT EQUAL TO +100 AND
  SQLCODE IS NOT EQUAL TO +222 THEN
  PERFORM PRINT-RESULTS.
:
:
*****
* Branch back to fetch the previous row. *
*****
:
:
*****
* Close the cursor *
*****
CLOSE-THISEMP.
EXEC SQL
  CLOSE THISEMP
END-EXEC.

```

The following example shows how to update an entire rowset with a cursor.

```

*****
* Declare a rowset cursor to update the JOB *
* column of the EMP table. *
*****
EXEC SQL
  DECLARE EMPSET CURSOR
  WITH ROWSET POSITIONING FOR
  SELECT EMPNO, LASTNAME, WORKDEPT, JOB
  FROM DSN8A10.EMP
  WHERE WORKDEPT = 'D11'
  FOR UPDATE OF JOB
END-EXEC.
*****
* Open the cursor. *
*****

```

```

EXEC SQL
  OPEN EMPSET
END-EXEC.
*****
* Indicate what action to take when end-of-data *
* occurs in the rowset being fetched.          *
*****
EXEC SQL
  WHENEVER NOT FOUND
  GO TO CLOSE-EMPSET
END-EXEC.
*****
* Fetch next rowset to position the cursor.    *
*****
EXEC SQL
  FETCH NEXT ROWSET FROM EMPSET
  FOR :SIZE-ROWSET ROWS
  INTO :HVA-EMPNO, :HVA-LASTNAME,
       :HVA-WORKDEPT, :HVA-JOB
END-EXEC.
*****
* Update rowset where the cursor is positioned. *
*****
UPDATE-ROWSET.
EXEC SQL
  UPDATE DSN8A10.EMP
  SET JOB = :NEW-JOB
  WHERE CURRENT OF EMPSET
END-EXEC.
END-UPDATE-ROWSET.
:
*****
* Branch back to fetch the next rowset.        *
*****
:
*****
* Update the remaining rows in the current *
* rowset and close the cursor.             *
*****
CLOSE-EMPSET.
PERFORM UPDATE-ROWSET.
EXEC SQL
  CLOSE EMPSET
END-EXEC.

```

The following example shows how to update specific rows with a rowset cursor.

```

*****
* Declare a static scrollable rowset cursor.    *
*****
EXEC SQL
  DECLARE EMPSET SENSITIVE STATIC SCROLL CURSOR
  WITH ROWSET POSITIONING FOR
  SELECT EMPNO, WORKDEPT, JOB
  FROM DSN8A10.EMP
  FOR UPDATE OF JOB
END-EXEC.
*****
* Open the cursor.                             *
*****
EXEC SQL
  OPEN EMPSET
END-EXEC.
*****
* Fetch next rowset to position the cursor.    *
*****
EXEC SQL

```

```

    FETCH SENSITIVE NEXT ROWSET FROM EMPSET
    FOR :SIZE-ROWSET ROWS
    INTO :HVA-EMPNO,
        :HVA-WORKDEPT :INDA-WORKDEPT,
        :HVA-JOB :INDA-JOB
END-EXEC.
*****
* Process fetch results if no error and no hole. *
*****
IF SQLCODE >= 0
    EXEC SQL GET DIAGNOSTICS
        :HV-ROWCNT = ROW_COUNT
    END-EXEC
    PERFORM VARYING N FROM 1 BY 1 UNTIL N > HV-ROWCNT
        IF INDA-WORKDEPT(N) NOT = -3
            EVALUATE HVA-WORKDEPT(N)
                WHEN ('D11')
                    PERFORM UPDATE-ROW
                WHEN ('E11')
                    PERFORM DELETE-ROW
            END-EVALUATE
        END-IF
    END-PERFORM
    IF SQLCODE = 100
        GO TO CLOSE-EMPSET
    END-IF
ELSE
    EXEC SQL GET DIAGNOSTICS
        :HV-NUMCOND = NUMBER
    END-EXEC
    PERFORM VARYING N FROM 1 BY 1 UNTIL N > HV-NUMCOND
        EXEC SQL GET DIAGNOSTICS CONDITION :N
            :HV-SQLCODE = DB2_RETURNED_SQLCODE,
            :HV-ROWNUM = DB2_ROW_NUMBER
        END-EXEC
        DISPLAY "SQLCODE = " HV-SQLCODE
        DISPLAY "ROW NUMBER = " HV-ROWNUM
    END-PERFORM
    GO TO CLOSE-EMPSET
END-IF.
:
:
*****
* Branch back to fetch and process *
* the next rowset. *
*****
:
:
*****
* Update row N in current rowset. *
*****
UPDATE-ROW.
EXEC SQL
    UPDATE DSN8A10.EMP
    SET JOB = :NEW-JOB
    FOR CURSOR EMPSET FOR ROW :N OF ROWSET
END-EXEC.
END-UPDATE-ROW.
*****
* Delete row N in current rowset. *
*****
DELETE-ROW.
EXEC SQL
    DELETE FROM DSN8A10.EMP
    WHERE CURRENT OF EMPSET FOR ROW :N OF ROWSET
END-EXEC.
END-DELETE-ROW.

```

```

:
*****
* Close the cursor. *
*****
CLOSE-EMPSET.
EXEC SQL
  CLOSE EMPSET
END-EXEC.

```

Specifying direct row access by using row IDs

For some applications, you can use the value of a ROWID column to navigate directly to a row.

Introductory concepts:

ROWID data type (Introduction to DB2 for z/OS)

When you select a ROWID column, the value implicitly contains the location of the retrieved row. If you use the value from the ROWID column in the search condition of a subsequent query, DB2 can choose to navigate directly to that row.

Example: Suppose that an EMPLOYEE table is defined in the following way:

```

CREATE TABLE EMPLOYEE
  (EMP_ROWID  ROWID NOT NULL GENERATED ALWAYS,
   EMPNO     SMALLINT,
   NAME      CHAR(30),
   SALARY    DECIMAL(7,2),
   WORKDEPT  SMALLINT);

```

The following code uses the SELECT from INSERT statement to retrieve the value of the ROWID column from a new row that is inserted into the EMPLOYEE table. This value is then used to reference that row for the update of the SALARY column.

```

EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS ROWID hv_emp_rowid;
  short             hv_dept, hv_empno;
  char              hv_name[30];
  decimal(7,2)     hv_salary;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
  SELECT EMP_ROWID INTO :hv_emp_rowid
  FROM FINAL TABLE (INSERT INTO EMPLOYEE
                    VALUES (DEFAULT, :hv_empno, :hv_name, :hv_salary, :hv_dept));
EXEC SQL
  UPDATE EMPLOYEE
  SET SALARY = SALARY + 1200
  WHERE EMP_ROWID = :hv_emp_rowid;

EXEC SQL COMMIT;

```

For DB2 to be able to use direct row access for the update operation, the SELECT from INSERT statement and the UPDATE statement must execute within the same unit of work. If these statements execute in different units of work, the ROWID value for the inserted row might change due to a REORG of the table space before the update operation. Alternatively, you can use a SELECT from MERGE statement. The MERGE statement performs INSERT and UPDATE operations as one coordinated statement.

ROWID columns as keys:

If you define a column in a table to have the ROWID data type, DB2 provides a unique value for each row in the table only if you define the column as GENERATED ALWAYS. The purpose of the value in the ROWID column is to uniquely identify rows in the table.

You can use a ROWID column to write queries that navigate directly to a row, which can be useful in situations where high performance is a requirement. This direct navigation, without using an index or scanning the table space, is called direct row access. In addition, a ROWID column is a requirement for tables that contain LOB columns. This topic discusses the use of a ROWID column in direct row access.

Requirement: To use direct row access, you must use a retrieved ROWID value before you commit. When your application commits, it releases its claim on the table space. After the commit, a REORG on your table space might execute and change the physical location of the rows.

Restriction: In general, you cannot use a ROWID column as a key that is to be used as a single column value across multiple tables. The ROWID value for a particular row in a table might change over time due to a REORG of the table space. In particular, you cannot use a ROWID column as part of a parent key or foreign key.

The value that you retrieve from a ROWID column is a varying-length character value that is not monotonically ascending or descending (the value is not always increasing or not always decreasing). Therefore, a ROWID column does not provide suitable values for many types of entity keys, such as order numbers or employee numbers.

Specifying direct row access by using RIDs:

When you specify a particular row ID, or RID, DB2 can navigate directly to the specified row for those queries that qualify for direct row access.

Before you begin this task, ensure that the query qualifies for direct row access. To qualify, the search condition must be a Boolean term, stage 1 predicate that fits one of the following criteria:

- A simple Boolean term predicate of the following form:

RID (table designator) = noncolumn expression

Where the noncolumn expression contains a result of a RID function.

- A compound Boolean term that combines several simple predicates by using the AND operator, where one of the simple predicates fits the first criteria.

To specify direct row access by using RIDs, specify the RID function in the search condition of a SELECT, DELETE, or UPDATE statement.

The RID function returns the RID of a row, which you can use to uniquely identify a row.

Restriction: Because DB2 might reuse RID numbers when the REORG utility is run, the RID function might return different values when invoked for a row multiple times.

If you specify a RID and DB2 cannot locate the row through direct row access, DB2 does not switch to another access method. Instead, DB2 returns no rows.

Related concepts:

➡ Direct row access (PRIMARY_ACCESTYPE='D') (DB2 Performance)

➡ Row ID values (DB2 SQL)

Related reference:

➡ RID (DB2 SQL)

ROWID columns

A *ROWID column* uniquely identifies each row in a table. This column enables queries to be written that navigate directly to a row in the table because the column implicitly contains the location of the row.

You can define a ROWID column as either GENERATED BY DEFAULT or GENERATED ALWAYS:

- If you define the column as GENERATED BY DEFAULT, you can insert a value. DB2 provides a default value if you do not supply one. However, to be able to insert an explicit value (by using the INSERT statement with the VALUES clause), you must create a unique index on that column.
- If you define the column as GENERATED ALWAYS (which is the default), DB2 always generates a unique value for the column. You cannot insert data into that column. In this case, DB2 does not require an index to guarantee unique values.

Related concepts:

“Rules for inserting data into a ROWID column” on page 658

➡ Direct row access (PRIMARY_ACCESTYPE='D') (DB2 Performance)

➡ ROWID data type (Introduction to DB2 for z/OS)

Related tasks:

➡ Specifying direct row access by using row IDs (DB2 Application programming and SQL)

Ways to manipulate LOB data

You can use SQL statements, LOB locators, and LOB file reference variables in your application programs to manipulate LOB data that is stored in DB2.

For example, you can use the following statements to extract information about an employee's department from the resume:

```
EXEC SQL BEGIN DECLARE SECTION;
    char    employeenum[6];
    long    deptInfoBeginLoc;
    long    deptInfoEndLoc;
    SQL TYPE IS CLOB_LOCATOR resume;
    SQL TYPE IS CLOB_LOCATOR deptBuffer;
EXEC SQL END DECLARE SECTION;
:
:
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT EMPNO, EMP_RESUME FROM EMP;
:
:
EXEC SQL FETCH C1 INTO :employeenum, :resume;
:
:
EXEC SQL SET :deptInfoBeginLoc =
    POSSTR(:resume.data, 'Department Information');
```

```
EXEC SQL SET :deptInfoEndLoc =
  POSSTR(:resume.data, 'Education');
```

```
EXEC SQL SET :deptBuffer =
  SUBSTR(:resume, :deptInfoBeginLoc,
  :deptInfoEndLoc - :deptInfoBeginLoc);
```

These statements use host variables of data type large object locator (LOB locator). LOB locators let you manipulate LOB data without moving the LOB data into host variables. By using LOB locators, you need much smaller amounts of memory for your programs.

You can also use LOB file reference variables when you are working with LOB data. You can use LOB file reference variables to insert LOB data from a file into a DB2 table or to retrieve LOB data from a DB2 table.


Sample LOB applications: The following table lists the sample programs that DB2 provides to assist you in writing applications to manipulate LOB data. All programs reside in data set DSNA10.SDSNSAMP.

Table 123. LOB samples shipped with DB2

Member that contains source code	Language	Function
DSNTEJ7	JCL	Demonstrates how to create a table with LOB columns, an auxiliary table, and an auxiliary index. Also demonstrates how to load LOB data that is 32 KB or less into a LOB table space.
DSN8DLPL	C	Demonstrates the use of LOB locators and UPDATE statements to move binary data into a column of type BLOB.
DSN8DLRV	C	Demonstrates how to use a locator to manipulate data of type CLOB.
DSNTEP2	PL/I	Demonstrates how to allocate an SQLDA for rows that include LOB data and use that SQLDA to describe an input statement and fetch data from LOB columns.

Related concepts:

“LOB file reference variables” on page 767

 Phase 7: Accessing LOB data (DB2 Installation and Migration)

Related tasks:

“Saving storage when manipulating LOBs by using LOB locators” on page 763

LOB host variable, LOB locator, and LOB file reference variable declarations

When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

You can declare LOB host variables and LOB locators in assembler, C, C++, COBOL, Fortran, and PL/I. Additionally, you can declare LOB file reference variables in assembler, C, C++, COBOL, and PL/I. REXX does not support LOB host variable, LOB locators, or LOB file reference variables. For each host variable,

locator, or file reference variable of SQL type BLOB, CLOB, or DBCLOB that you declare, DB2 generates an equivalent declaration that uses host language data types. When you refer to a LOB host variable, LOB locator, or LOB file reference variable in an SQL statement, you must use the variable that you specified in the SQL type declaration. When you refer to the host variable in a host language statement, you must use the variable that DB2 generates.

DB2 supports host variable declarations for LOBs with lengths of up to 2 GB - 1. However, the size of a LOB host variable is limited by the restrictions of the host language and the amount of storage available to the program.

Declare LOB host variables that are referenced by the precompiler in SQL statements by using the SQL TYPE IS BLOB, SQL TYPE IS CLOB, or SQL TYPE IS DBCLOB keywords.

LOB host variables that are referenced only by an SQL statement that uses a DESCRIPTOR should use the same form as declared by the precompiler. In this form, the LOB host-variable-array consists of a 31-bit length, followed by the data, followed by another 31-bit length, followed by the data, and so on. The 31-bit length must be fullword aligned.

Example: Suppose that you want to allocate a LOB array of 10 elements, each with a length of 5 bytes. You need to allocate the following bytes for each element, for a total of 120 bytes:

- 4 bytes for the 31-bit integer
- 5 bytes for the data
- 3 bytes to force fullword alignment

The following examples show you how to declare LOB host variables in each supported language. In each table, the left column contains the declaration that you code in your application program. The right column contains the declaration that DB2 generates.

Declarations of LOB host variables in assembler

The following table shows assembler language declarations for some typical LOB types.

Table 124. Example of assembler LOB variable declarations

You declare this variable	DB2 generates this variable
clob_var SQL TYPE IS CLOB 40000K	clob_var DS 0FL4 clob_var_length DS FL4 clob_var_data DS CL65535 ¹ ORG clob_var_data +(40960000-65535)
dbclob_var SQL TYPE IS DBCLOB 4000K	dbclob_var DS 0FL4 dbclob_var_length DS FL4 dbclob_var_data DS GL65534 ² ORG dbclob_var_data+(8192000-65534)
blob_var SQL TYPE IS BLOB 1M	blob_var DS 0FL4 blob_var_length DS FL4 blob_var_data DS CL65535 ¹ ORG blob_var_data+(1048476-65535)
clob_loc SQL TYPE IS CLOB_LOCATOR	clob_loc DS FL4
dbclob_loc SQL TYPE IS DBCLOB_LOCATOR	dbclob_loc DS FL4

Table 124. Example of assembler LOB variable declarations (continued)

You declare this variable	DB2 generates this variable
blob_loc SQL TYPE IS BLOB_LOCATOR	blob_loc DS FL4
clob_file SQL TYPE IS CLOB_FILE	clob_file DS FL4
dbclob_file SQL TYPE IS DBCLOB_FILE	dbclob_file DS FL4
blob_file SQL TYPE IS BLOB_FILE	blob_file DS FL4

Notes:

1. Because assembler language allows character declarations of no more than 65535 bytes, DB2 separates the host language declarations for BLOB and CLOB host variables that are longer than 65535 bytes into two parts.
2. Because assembler language allows graphic declarations of no more than 65534 bytes, DB2 separates the host language declarations for DBCLOB host variables that are longer than 65534 bytes into two parts.

Declarations of LOB host variables in C

The following table shows C and C++ language declarations for some typical LOB types.

Table 125. Examples of C language variable declarations

You declare this variable	DB2 generates this variable
SQL TYPE IS BLOB (1M) blob_var;	struct { unsigned long length; char data[1048576]; } blob_var;
SQL TYPE IS CLOB(400K) clob_var;	struct { unsigned long length; char data[409600]; } clob_var;
SQL TYPE IS DBCLOB (4000K) dbclob_var;	struct { unsigned long length; sqlbchar data[4096000]; } dbclob_var;
SQL TYPE IS BLOB_LOCATOR blob_loc;	unsigned long blob_loc;
SQL TYPE IS CLOB_LOCATOR clob_loc;	unsigned long clob_loc;
SQL TYPE IS DBCLOB_LOCATOR dbclob_loc;	unsigned long dbclob_loc;
SQL TYPE IS BLOB_FILE FBLOBhv;	#pragma pack(full) struct { unsigned long name_length; unsigned long data_length; unsigned long file_options; char name??(255??); } FBLOBhv ; #pragma pack(reset)
SQL TYPE IS CLOB_FILE FCLOBhv;	#pragma pack(full) struct { unsigned long name_length; unsigned long data_length; unsigned long file_options; char name??(255??); } FCLOBhv ; #pragma pack(reset)

Table 125. Examples of C language variable declarations (continued)

You declare this variable	DB2 generates this variable
SQL TYPE IS DBCLOB_FILE FDBCLOBhv;	#pragma pack(full) struct { unsigned long name_length; unsigned long data_length; unsigned long file_options; char name??(255??); } FDBCLOBhv ; #pragma pack(reset)

Declarations of LOB host variables in COBOL

The declarations that are generated for COBOL depend on whether you use the DB2 precompiler or the DB2 coprocessor. The following table shows COBOL declarations that the DB2 precompiler generates for some typical LOB types. The declarations that the DB2 coprocessor generates might be different.

Table 126. Examples of COBOL variable declarations by the DB2 precompiler

You declare this variable	DB2 precompiler generates this variable
01 BLOB-VAR SQL TYPE IS BLOB(1M).	01 BLOB-VAR. 49 BLOB-VAR-LENGTH PIC S9(9) COMP-5. 49 BLOB-VAR-DATA PIC X(1048576).
01 CLOB-VAR SQL TYPE IS CLOB(40000K).	01 CLOB-VAR. 49 CLOB-VAR-LENGTH PIC S9(9) COMP-5. 49 CLOB-VAR-DATA PIC X(40960000).
01 DBCLOB-VAR SQL TYPE IS DBCLOB(4000K).	01 DBCLOB-VAR. 49 DBCLOB-VAR-LENGTH PIC S9(9) COMP-5 49 DBCLOB-VAR-DATA PIC G(40960000) DISPLAY-1.
01 BLOB-LOC SQL TYPE IS BLOB-LOCATOR.	01 BLOB-LOC PIC S9(9) COMP-5.
01 CLOB-LOC SQLTYPE IS CLOB-LOCATOR.	01 CLOB-LOC PIC S9(9) COMP-5.
01 DBCLOB-LOC SQLTYPE IS DBCLOB-LOCATOR.	01 DBCLOB-LOC PIC S9(9) COMP-5.
01 BLOB-FILE SQLTYPE IS BLOB-FILE.	01 BLOB-FILE. 49 BLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 BLOB-FILE-DATA-LENGTH PIC S9(9) COMP-5. 49 BLOB-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 BLOB-FILE-NAME PIC X(255) .
01 CLOB-FILE SQLTYPE IS CLOB-FILE.	01 CLOB-FILE. 49 CLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 CLOB-FILE-DATA-LENGTH PIC S9(9) COMP-5. 49 CLOB-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 CLOB-FILE-NAME PIC X(255) .
01 DBCLOB-FILE SQLTYPE IS DBCLOB-FILE.	01 DBCLOB-FILE. 49 DBCLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 DBCLOB-FILE-DATA-LENGTH PIC S9(9) COMP-5. 49 DBCLOB-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 DBCLOB-FILE-NAME PIC X(255) .

Declarations of LOB host variables in Fortran

The following table shows Fortran declarations for some typical LOB types.

Table 127. Examples of Fortran variable declarations

You declare this variable	DB2 generates this variable
SQL TYPE IS BLOB(1M) blob_var	CHARACTER blob_var(1048580) INTEGER*4 blob_var_LENGTH CHARACTER blob_var_DATA EQUIVALENCE(blob_var(1), + blob_var_LENGTH) EQUIVALENCE(blob_var(5), + blob_var_DATA)
SQL TYPE IS CLOB(40000K) clob_var	CHARACTER clob_var(4096004) INTEGER*4 clob_var_length CHARACTER clob_var_data EQUIVALENCE(clob_var(1), + clob_var_length) EQUIVALENCE(clob_var(5), + clob_var_data)
SQL TYPE IS BLOB_LOCATOR blob_loc	INTEGER*4 blob_loc
SQL TYPE IS CLOB_LOCATOR clob_loc	INTEGER*4 clob_loc

Declarations of LOB host variables in PL/I

The declarations that are generated for PL/I depend on whether you use the DB2 precompiler or the DB2 coprocessor. The following table shows PL/I declarations that the DB2 precompiler generates for some typical LOB types. The declarations that the DB2 coprocessor generates might be different.

Table 128. Examples of PL/I variable declarations by the DB2 precompiler

You declare this variable	DB2 precompiler generates this variable
DCL BLOB_VAR SQL TYPE IS BLOB (1M);	DCL 1 BLOB_VAR, 2 BLOB_VAR_LENGTH FIXED BINARY(31), 2 BLOB_VAR_DATA, ¹ 3 BLOB_VAR_DATA1(32) CHARACTER(32767), 3 BLOB_VAR_DATA2 CHARACTER(1048576-32*32767);
DCL CLOB_VAR SQL TYPE IS CLOB (40000K);	DCL 1 CLOB_VAR, 2 CLOB_VAR_LENGTH FIXED BINARY(31), 2 CLOB_VAR_DATA, ¹ 3 CLOB_VAR_DATA1(1250) CHARACTER(32767), 3 CLOB_VAR_DATA2 CHARACTER(4096000-1250*32767);
DCL DBCLOB_VAR SQL TYPE IS DBCLOB (4000K);	DCL 1 DBCLOB_VAR, 2 DBCLOB_VAR_LENGTH FIXED BINARY(31), 2 DBCLOB_VAR_DATA, ² 3 DBCLOB_VAR_DATA1(250) GRAPHIC(16383), 3 DBCLOB_VAR_DATA2 GRAPHIC(4096000-250*16383);
DCL blob_loc SQL TYPE IS BLOB_LOCATOR;	DCL blob_loc FIXED BINARY(31);

Table 128. Examples of PL/I variable declarations by the DB2 precompiler (continued)

You declare this variable	DB2 precompiler generates this variable
DCL clob_loc SQL TYPE IS CLOB_LOCATOR;	DCL clob_loc FIXED BINARY(31);
DCL dbclob_loc SQL TYPE IS DBCLOB_LOCATOR;	DCL dbclob_loc FIXED BINARY(31);
DCL blob_file SQL TYPE IS BLOB_FILE;	DCL 1 blob_file, 2 blob_file_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 blob_file_DATA_LENGTH BIN FIXED(31), 2 blob_file_FILE_OPTIONS BIN FIXED(31), 2 blob_file_NAME CHAR(255) ;
DCL clob_file SQL TYPE IS CLOB_FILE;	DCL 1 clob_file, 2 clob_file_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 clob_file_DATA_LENGTH BIN FIXED(31), 2 clob_file_FILE_OPTIONS BIN FIXED(31), 2 clob_file_NAME CHAR(255) ;
DCL dbclob_file SQL TYPE IS DBCLOB_FILE;	DCL 1 dbclob_file, 2 dbclob_file_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 dbclob_file_DATA_LENGTH BIN FIXED(31), 2 dbclob_file_FILE_OPTIONS BIN FIXED(31), 2 dbclob_file_NAME CHAR(255) ;

Notes:

- For BLOB or CLOB host variables that are greater than 32767 bytes in length, DB2 creates PL/I host language declarations in the following way:
 - If the length of the LOB is greater than 32767 bytes and evenly divisible by 32767, DB2 creates an array of 32767-byte strings. The dimension of the array is $length/32767$.
 - If the length of the LOB is greater than 32767 bytes but not evenly divisible by 32767, DB2 creates two declarations: The first is an array of 32767 byte strings, where the dimension of the array, n , is $length/32767$. The second is a character string of length $length-n*32767$.
- For DBCLOB host variables that are greater than 16383 double-byte characters in length, DB2 creates PL/I host language declarations in the following way:
 - If the length of the LOB is greater than 16383 characters and evenly divisible by 16383, DB2 creates an array of 16383-character strings. The dimension of the array is $length/16383$.
 - If the length of the LOB is greater than 16383 characters but not evenly divisible by 16383, DB2 creates two declarations: The first is an array of 16383 byte strings, where the dimension of the array, m , is $length/16383$. The second is a character string of length $length-m*16383$.

Related concepts:

“LOB file reference variables” on page 767

Related tasks:

“Saving storage when manipulating LOBs by using LOB locators” on page 763

LOB and XML materialization

Materialization means that DB2 puts the data that is selected into a buffer for processing. This action can slow performance. Because LOB values can be very large, DB2 avoids materializing LOB data until absolutely necessary.

Beginning in Version 10, LOB and XML materialization has been reduced or eliminated within DB2 for several local and distributed cases including utilities (LOAD and cross-loader). Some of the cases where materialization has been

eliminated or reduced include during DRDA streaming, file reference variable processing, CCSID conversion and distributed XML fetch processing. However, whether the values will be materialized and how much will be materialized also depends on the number and size of each LOB or XML.

DB2 stores LOB values in contiguous storage. DB2 must materialize LOBs when your application program performs the following actions:

- Calls a user-defined function with a LOB as an argument
- Moves a LOB into or out of a stored procedure
- Assigns a LOB host variable to a LOB locator host variable

The amount of storage that is used for LOB and XML materialization depends on a number of factors including:

- The size of the LOBs
- The number of LOBs that need to be materialized in a statement

DB2 loads LOBs into virtual pools above the bar. If insufficient space is available for LOB materialization, your application receives SQLCODE -904.

Although you cannot completely avoid LOB materialization, you can minimize it by using LOB locators, rather than LOB host variables in your application programs.

Related tasks:

“Saving storage when manipulating LOBs by using LOB locators”

Saving storage when manipulating LOBs by using LOB locators

LOB locators let you manipulate LOB data without retrieving the data from the DB2 table. By using locators, you avoid needing to allocate the large amounts of storage that are needed for host variables to hold LOB data.

To retrieve LOB data from a DB2 table, you can define host variables that are large enough to hold all of the LOB data. This requires your application to allocate large amounts of storage, and requires DB2 to move large amounts of data, which can be inefficient or impractical. Instead, you can use LOB locators. LOB locators let you manipulate LOB data without retrieving the data from the DB2 table. Using LOB locators for LOB data retrieval is a good choice in the following situations:

- When you move only a small part of a LOB to a client program
- When the entire LOB does not fit in the application's memory
- When the program needs a temporary LOB value from a LOB expression but does not need to save the result
- When performance is important

A LOB locator is associated with a LOB value or expression, not with a row in a DB2 table or a physical storage location in a table space. Therefore, after you select a LOB value using a locator, the value in the locator normally does not change until the current unit of work ends. However the value of the LOB itself can change.

If you want to remove the association between a LOB locator and its value before a unit of work ends, execute the FREE LOCATOR statement. To keep the association between a LOB locator and its value after the unit of work ends, execute the HOLD LOCATOR statement. After you execute a HOLD LOCATOR statement, the

locator keeps the association with the corresponding value until you execute a `FREE LOCATOR` statement or the program ends.

If you execute `HOLD LOCATOR` or `FREE LOCATOR` dynamically, you cannot use `EXECUTE IMMEDIATE`.

Applications that use a huge number of locators, which commit infrequently, or do not explicitly free the locators, can use large amounts of valuable DBM1 storage and CPU costs. Frequently use `COMMIT` or `FREE LOCATORS` to avoid storage shortage on the DBM1 address space and a shortage of system CPU resource.

To free LOB locators after their associated LOB values are retrieved, run the `FREE LOCATOR` statement:

```
EXEC SQL FREE LOCATOR :LOCRES, :LOCHIST, :LOCPIC
```

Related reference:

[FREE LOCATOR \(DB2 SQL\)](#)

[HOLD LOCATOR \(DB2 SQL\)](#)

Indicator variables and LOB locators

DB2 uses indicator variables for LOB locators differently than it uses indicator variables for host variables.

For host variables other than LOB locators, when you select a null value into a host variable, DB2 assigns a negative value to the associated indicator variable. However, for LOB locators, DB2 uses indicator variables differently. A LOB locator is never null. When you select a LOB column using a LOB locator and the LOB column contains a null value, DB2 assigns a null value to the associated indicator variable. The value in the LOB locator does not change. In a client/server environment, this null information is recorded only at the client.

When you use LOB locators to retrieve data from columns that can contain null values, define indicator variables for the LOB locators, and check the indicator variables after you fetch data into the LOB locators. If an indicator variable is null after a fetch operation, you cannot use the value in the LOB locator.

Valid assignments for LOB locators

Although you usually use LOB locators to assign data to and retrieve data from LOB columns, you can also use LOB locators to assign data to non-LOB columns.

You can use LOB locators to make the following assignments:

- A CLOB or DBCLOB locator can be assigned to a `CHAR`, `VARCHAR`, `GRAPHIC`, or `VARGRAPHIC` column. However, you cannot fetch data from `CHAR`, `VARCHAR`, `GRAPHIC`, or `VARGRAPHIC` columns into a CLOB or DBCLOB locators.
- A BLOB locator can be assigned to a `BINARY` or `VARBINARY` column. However, you cannot fetch data from a `BINARY` or `VARBINARY` column into a BLOB locator.

Avoiding character conversion for LOB locators

In certain situations, DB2 materializes the entire LOB value and converts it to the encoding scheme of a particular SQL statement. This extra processing can degrade performance and should be avoided.

You can use a VALUES INTO or SET statement to obtain the results of functions that operate on LOB locators, such as LENGTH or SUBSTR. VALUES INTO and SET statements are processed in the application encoding scheme for the plan or package that contains the statement. If that encoding scheme is different from the encoding scheme of the LOB data, the entire LOB value is materialized and converted to the encoding scheme of the statement. This materialization and conversion processing can cause performance degradation.

To avoid the character conversion, SELECT from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table. These dummy tables perform functions similar to SYSIBM.SYSDUMMY1, and are each associated with an encoding scheme:

SYSIBM.SYSDUMMYA

ASCII

SYSIBM.SYSDUMMYE

EBCDIC

SYSIBM.SYSDUMMYU

Unicode

By using these tables, you can obtain the same result as you would with a VALUES INTO or SET statement.

Suppose that the encoding scheme of the following statement is EBCDIC:

```
SET : unicode_hv = SUBSTR(:Unicode_lob_locator,X,Y);
```

DB2 must materialize the LOB that is specified by :Unicode_lob_locator and convert that entire LOB to EBCDIC before executing the statement. To avoid materialization and conversion, you can execute the following statement, which produces the same result but is processed by the Unicode encoding scheme of the table:

```
SELECT SUBSTR(:Unicode_lob_locator,X,Y) INTO :unicode_hv  
FROM SYSIBM.SYSDUMMYU;
```

Deferring evaluation of a LOB expression to improve performance

DB2 does not move any bytes of a LOB value until a program assigns a LOB expression to a target destination. When you use a LOB locator with string functions and operators, DB2 does not evaluate the expression until the time of assignment. This deferred evaluation can improve performance.

The following example is a C language program that defers evaluation of a LOB expression. The program runs on a client and modifies LOB data at a server. The program searches for a particular resume (EMPNO = '000130') in the EMP_RESUME table. It then uses LOB locators to rearrange a copy of the resume (with EMPNO = 'A00130'). In the copy, the Department Information Section appears at the end of the resume. The program then inserts the copy into EMP_RESUME without modifying the original resume.

Because the program in the following figure uses LOB locators, rather than placing the LOB data into host variables, no LOB data is moved until the INSERT statement executes. In addition, no LOB data moves between the client and the server.

```

EXEC SQL INCLUDE SQLCA;

/*****/
/* Declare host variables */
/*****/
EXEC SQL BEGIN DECLARE SECTION;
    char userid[9];
    char passwd[19];
    long      HV_START_DEPTINFO;
    long      HV_START_EDUC;
    long      HV_RETURN_CODE;
    SQL TYPE IS CLOB_LOCATOR HV_NEW_SECTION_LOCATOR;
    SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR1;
    SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR2;
    SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR3;
EXEC SQL END DECLARE SECTION;

/*****/
/* Delete any instance of "A00130" from previous */
/* executions of this sample */
/*****/
EXEC SQL DELETE FROM EMP_RESUME WHERE EMPNO = 'A00130';

/*****/
/* Use a single row select to get the document */
/*****/
EXEC SQL SELECT RESUME
        INTO :HV_DOC_LOCATOR1
        FROM EMP_RESUME
        WHERE EMPNO = '000130'
        AND RESUME_FORMAT = 'ascii';

/*****/
/* Use the POSSTR function to locate the start of */
/* sections "Department Information" and "Education" */
/*****/
EXEC SQL SET :HV_START_DEPTINFO =
        POSSTR(:HV_DOC_LOCATOR1, 'Department Information');

EXEC SQL SET :HV_START_EDUC =
        POSSTR(:HV_DOC_LOCATOR1, 'Education');

/*****/
/* Replace Department Information section with nothing */
/*****/
EXEC SQL SET :HV_DOC_LOCATOR2 =
        SUBSTR(:HV_DOC_LOCATOR1, 1, :HV_START_DEPTINFO -1)
        || SUBSTR (:HV_DOC_LOCATOR1, :HV_START_EDUC);

/*****/
/* Associate a new locator with the Department */
/* Information section */
/*****/
EXEC SQL SET :HV_NEW_SECTION_LOCATOR =
        SUBSTR(:HV_DOC_LOCATOR1, :HV_START_DEPTINFO,
        :HV_START_EDUC - :HV_START_DEPTINFO);

/*****/
/* Append the Department Information to the end */
/* of the resume */
/*****/
EXEC SQL SET :HV_DOC_LOCATOR3 =
        :HV_DOC_LOCATOR2 || :HV_NEW_SECTION_LOCATOR;

/*****/
/* Store the modified resume in the table. This is */
/* where the LOB data really moves. */
/*****/
EXEC SQL INSERT INTO EMP_RESUME VALUES ('A00130', 'ascii',
        :HV_DOC_LOCATOR3, DEFAULT);

```

1

2

3

4


```

/*****/
/* Free the locators */
/*****/
EXEC SQL FREE LOCATOR :HV_DOC_LOCATOR1, :HV_DOC_LOCATOR2, :HV_DOC_LOCATOR3;

```

Notes:

- 1** Declare the LOB locators here.
- 2** This SELECT statement associates LOB locator HV_DOC_LOCATOR1 with the value of column RESUME for employee number 000130.
- 3** The next five SQL statements use LOB locators to manipulate the resume data without moving the data.
- 4** Evaluation of the LOB expressions in the previous statements has been deferred until execution of this INSERT statement.
- 5** Free all LOB locators to release them from their associated values.

LOB file reference variables

In a host application, you can use a file reference variable to insert a LOB or XML value from a file into a DB2 table. You can also use a file reference variable to select a LOB or XML value from a DB2 table into a file.

The file reference variables are BLOB_FILE, CLOB_FILE, or DBCLOB_FILE. For COBOL, the file reference variables are BLOB-FILE, CLOB-FILE, or DBCLOB-FILE.

When you use a file reference variable, you can select or insert an entire LOB or XML value without contiguous application storage to contain the entire LOB or XML value. LOB file reference variables move LOB or XML values from the database server to an application or from an application to the database server without going through the application's memory. Furthermore, LOB file reference variables bypass the host language limitation on the maximum size allowed for dynamic storage to contain a LOB value.

You can declare LOB or XML values as LOB file reference variables or LOB file reference arrays for applications that are written in C, COBOL, PL/I, and assembler. The LOB file reference variables do not contain LOB data; they represent a file that contains LOB data. Database queries, updates, and inserts can use file reference variables to store or retrieve column values. As with other host variables, a LOB file reference variable can have an associated indicator variable.

DB2-generated LOB file reference variable constructs

For each LOB file reference variable that an application declares for a LOB or XML value, DB2 generates an equivalent construct that uses the host language data types. When an application references a LOB file reference variable, it must use the equivalent construct that DB2 generates; otherwise the DB2 precompiler issues an error.

The construct describes the following properties of the file:

Data type

BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared by using the BLOB_FILE, CLOB_FILE, or DBCLOB_FILE data type.

For COBOL, the data types are BLOB-FILE, CLOB-FILE, or DBCLOB-FILE.

Direction

This property must be specified by the application program at run time as part of the file option property. The direction property can have the following values:

Input Used as a data source on an EXECUTE, OPEN, UPDATE, INSERT, DELETE, SET, or MERGE statement.

Output

Used as the target of data on a FETCH statement or a SELECT INTO statement.

File name

This property must be specified by the application program at run time. The file name property can have the following values:

- The complete path name of the file. This is recommended.

File name length

This property must be specified by the application program at run time.

File options

An application must assign one of the file options to a file reference variable before the application can use that variable. File options are set by the INTEGER value in a field in the file reference variable construct. One of the following values must be specified for each file reference variable:

- Input (from application to database):

SQL_FILE_READ

A regular file that can be opened, read, and closed.

- Output (from database to application):

SQL_FILE_CREATE

If the file does not exist, a new file is created. If the file already exists, an error is returned.

SQL_FILE_OVERWRITE

If the file does not exist, a new file is created. If the file already exists, it is overwritten.

SQL_FILE_APPEND

If the file does not exist, a new file is created. If the file already exists, the output is appended to the existing file.

Data length

The length, in bytes, of the new data written to the file

Examples of declaring file reference variables

You can declare a file reference variable in C, COBOL, and PL/I, and declare the file reference variable construct that DB2 generates.

C Example: Consider the following C declaration:

```
EXEC SQL BEGIN DECLARE SECTION
  SQL TYPE IS CLOB_FILE hv_text_file;
  CHAR hv_thesis_title[64];
EXEC SQL END DECLARE SECTION
```

That declaration results in the following DB2-generated construct:

```
EXEC SQL BEGIN DECLARE SECTION
  /* SQL TYPE IS CLOB_FILE hv_text_file; */
  struct {
    unsigned long name_length; // File name length
```

```

        unsigned long data_length; // Data length
        unsigned long file_options; // File options
        char name [255]; // File name
    } hv_text_file;
char hv_thesis_title[64]

```

With the DB2-generated construct, you can use the following code to select from a CLOB column in the database into a new file that is referenced by :hv_text_file. The file name must be an absolute path.

```

strcpy(hv_text_file.name, "/u/gainer/papers/sigmod.94");
    hv_text_file.name_length = strlen("/u/gainer/papers/sigmod.94");
    hv_text_file.file_options = SQL_FILE_CREATE;
EXEC SQL SELECT CONTENT INTO :hv_text_file FROM PAPERS
        WHERE TITLE = 'The Relational Theory Behind Juggling';

```

Similarly, you can use the following code to insert the data from a file that is referenced by :hv_text_file into a CLOB column. The file name must be an absolute path.

```

strcpy(hv_text_file.name, "/u/gainer/patents/chips.13");
    hv_text_file.name_length = strlen("/u/gainer/patents/chips.13");
    hv_text_file.file_options = SQL_FILE_READ;
strcpy(:hv_patent_title, "A Method for Pipelining Chip Consumption");
EXEC SQL INSERT INTO PATENTS(TITLE, TEXT)
        VALUES(:hv_patent_title, :hv_text_file);

```

COBOL Example: Consider the following COBOL declaration:

```
01 MY-FILE SQL TYPE IS BLOB-FILE
```

That declaration results in the following DB2-generated construct:

```

01 MY-FILE.
    49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
    49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
    49 MY-FILE-FILE-OPTION PIC S9(9) COMP-5.
    49 MY-FILE-NAME PIC(255);

```

PL/I Example: Consider the following PL/I declaration:

```
DCL MY_FILE SQL TYPE IS CLOB_FILE
```

That declaration results in the following DB2-generated construct:

```

DCL 1 MY_FILE,
    3 MY_FILE_NAME_LENGTH BINARY FIXED (31) UNALIGNED,
    3 MY_FILE_DATA_LENGTH BINARY FIXED (31) UNALIGNED,
    3 MY_FILE_FILE_OPTIONS BINARY FIXED (31) UNALIGNED,
    3 MY_FILE_NAME CHAR(255);

```

For examples of how to declare file reference variables for XML data in C, COBOL, and PL/I, see “Host variable data types for XML data in embedded SQL applications” on page 241.

Referencing a sequence object

A *sequence* object is a user-defined object that generates a sequence of numeric values according to the specification with which the sequence was created. You can retrieve the next or previous value in the sequence.

You reference a sequence by using the NEXT VALUE expression or the PREVIOUS VALUE expression, specifying the name of the sequence:

- A NEXT VALUE expression generates and returns the next value for the specified sequence. If a query contains multiple instances of a NEXT VALUE expression with the same sequence name, the sequence value increments only once for that query. The ROLLBACK statement has no effect on values already generated.
- A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous NEXT VALUE expression that specified the same sequence within the current application process. The value of the PREVIOUS VALUE expression persists until the next value is generated for the sequence, the sequence is dropped, or the application session ends. The COMMIT statement and the ROLLBACK statement have no effect on this value.

You can specify a NEXT VALUE or PREVIOUS VALUE expression in a SELECT clause, within a VALUES clause of an insert operation, within the SET clause of an update operation (with certain restrictions), or within a SET *host-variable* statement.

Retrieving thousands of rows

When retrieving large numbers of rows, consider the possibilities for lock escalation and other locking issues.

Question: Are there any special techniques for fetching and displaying large volumes of data?

Answer: There are no special techniques; but for large numbers of rows, efficiency can become very important. In particular, you need to be aware of locking considerations, including the possibilities of lock escalation.

If your program allows input from a terminal before it commits the data and thereby releases locks, it is possible that a significant loss of concurrency results.

Determining when a row was changed

If a table has a ROW CHANGE TIMESTAMP column, you can determine when a row was changed.

To determine when a row was changed:

Issue a SELECT statement with the ROW CHANGE TIMESTAMP column in the column list. If a qualifying row does not have a value for the ROW CHANGE TIMESTAMP column, DB2 returns the time that the page in which that row resides was updated.

Suppose that you issue the following statements to create, populate, and alter a table:

```
CREATE TABLE T1 (C1 INTEGER NOT NULL);
INSERT INTO T1 VALUES (1);
ALTER TABLE T1 ADD COLUMN C2 NOT NULL GENERATED ALWAYS
  FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP;
SELECT T1.C2 FROM T1 WHERE T1.C1 = 1;
```

Because the ROW CHANGE TIMESTAMP column was added after the data was inserted, the following statement returns the time that the page was last modified:

```
SELECT T1.C2 FROM T1 WHERE T1.C1 = 1;
```

Assume that you then issue the following statement:

```
INSERT INTO T1(C1) VALUES (2);
```

Assume that this row is added to the same page as the first row. The following statement returns the time that value "2" was inserted into the table:

```
SELECT T1.C2 FROM T1 WHERE T1.C1 = 2;
```

Because the row with value "1" still does not have a value for the ROW CHANGE TIMESTAMP column, the following statement still returns the time that the page was last modified, which in this case is the time that value "2" was inserted:

```
SELECT T1.C2 FROM T1 WHERE T1.C1 = 1;
```

Related reference:

 CREATE TABLE (DB2 SQL)

Checking whether an XML column contains a certain value

You can determine which rows contain any fragment of XML data that you specify.

To check whether an XML column contains a certain value:

Specify the XMLEXISTS predicate in the WHERE clause of your SQL statement. Include the following parameters for the XMLEXISTS predicate:

- An XPath expression that is embedded in a character string literal. Specify an XPath expression that identifies the XML data that you are looking for. If the result of the XPath expression is an empty sequence, XMLEXISTS returns false. If the result is not empty, XMLEXISTS returns true. If the evaluation of the XPath expression returns an error, XMLEXISTS returns an error.
- The XML column name. Specify this value after the PASSING keyword.

Suppose that you want to return only purchase orders that have a billing address. Assume that column XMLPO stores the XML purchase order documents and that the billTo nodes within these documents contain any billing addresses. You can use the following SELECT statement with the XMLEXISTS predicate:

```
SELECT XMLPO FROM T1
  WHERE XMLEXISTS ('declare namespace ipo="http://www.example.com/IP0";
                  /ipo:purchaseOrder[billTo]'
                  PASSING XMLPO);
```

Related reference:

 XMLEXISTS predicate (DB2 SQL)

Accessing DB2 data that is not in a table

You can access DB2 data that is not in a table by returning the value of an SQL expression in a host variable.

The expression does not include a column of a table. The three ways to return a value in a host variable are shown in the following examples.

Example: To set the contents of a host variable to the value of an expression, use the SET host-variable assignment statement:

```
EXEC SQL SET :hvranda1 = RAND(:hvranda);
```

Example: To return the value of an expression in a host variable, use the VALUES INTO statement:

```
EXEC SQL VALUES RAND(:hvrاند)
INTO :hvrاندval;
```

Example: To select the expression from the DB2-provided EBCDIC table, named SYSIBM.SYSDUMMY1, which consists of one row, use the following statement:

```
EXEC SQL SELECT RAND(:hvrاند)
INTO :hvrاندval
FROM SYSIBM.SYSDUMMY1;
```

Ensuring that queries perform sufficiently

It is important to make sure that any individual queries that are included in your program are not slowing down the performance of your program.

To ensure that queries perform sufficiently:

1. Tune each query in your program by following the general tuning guidelines for how to write efficient queries.
2. If you suspect that a query is not as efficient as it could be, monitor its performance. You can use a number of different functions and techniques to monitor SQL performance, including the SQL EXPLAIN statement and SQL optimization tools.

Related concepts:

[🔗 Investigating SQL performance by using EXPLAIN \(DB2 Performance\)](#)

[🔗 Interpreting data access by using EXPLAIN \(DB2 Performance\)](#)

Related tasks:

[🔗 Programming applications for performance \(DB2 Performance\)](#)

[🔗 Investigating access path problems \(DB2 Performance\)](#)

[🔗 Generating visual representations of access plans \(IBM Data Studio\)](#)

Related reference:

[🔗 EXPLAIN \(DB2 SQL\)](#)

[🔗 InfoSphere Optim Query Workload Tuner](#)

Related information:

[🔗 Tuning SQL with Optim Query Tuner, Part 1: Understanding access paths \(IBM developerWorks\)](#)

Items to include in a batch DL/I program

When you use a batch DL/I program with DB2, you must include certain items in your program.

A batch DL/I program can issue:

- Any IMS batch call, except ROLS, SETS, and SYNC calls. ROLS and SETS calls provide intermediate backout point processing, which DB2 does not support. The SYNC call provides commit point processing without identifying the commit point with a value. IMS does not allow a SYNC call in batch, and neither does the DB2 DL/I batch support.

Issuing a ROLS, SETS, or SYNC call in an application program causes a system abend X'04E' with the reason code X'00D44057' in register 15.

- GSAM calls.

- IMS system services calls.
- Any SQL statements, except COMMIT and ROLLBACK. IMS and CICS environments do not allow those SQL statements; however, IMS and CICS do allow ROLLBACK TO SAVEPOINT. You can use the IMS CHKP call to commit data and the IMS ROLL or ROLB to roll back changes.
Issuing a COMMIT statement causes SQLCODE -925; issuing a ROLLBACK statement causes SQLCODE -926. Those statements also return SQLSTATE '2D521'.
- Any call to a standard or traditional access method (for example, QSAM, VSAM, and so on).

The restart capabilities for DB2 and IMS databases, as well as for sequential data sets that are accessed through GSAM, are available through the IMS Checkpoint and Restart facility.

DB2 allows access to both DB2 and DL/I data through the use of the following DB2 and IMS facilities:

- IMS synchronization calls, which commit and abnormally terminate units of recovery
- The DB2 IMS attachment facility, which handles the two-phase commit protocol and enables both systems to synchronize a unit of recovery during a restart after a failure
- The IMS log, which is used to record the instant of commit

In a data sharing environment, DL/I batch supports group attachment or subgroup attachment. You can specify a group attachment name instead of a subsystem name in the SSN parameter of the DDITV02 data set for the DL/I batch job.

Requirements for using DB2 in a DL/I batch job

Using DB2 in a DL/I batch job requires the following changes to the application program and the job step JCL:

- Add SQL statements to your application program to gain access to DB2 data. You must then precompile the application program and bind the resulting DBRM into a package.
- Before you run the application program, use JOBLIB, STEPLIB, or link book to access the DB2 load library, so that DB2 modules can be loaded.
- In a data set that is specified by a DDITV02 DD statement, specify the program name and plan name for the application, and the connection name for the DL/I batch job.

In an input data set or in a subsystem member, specify information about the connection between DB2 and IMS. The input data set name is specified with a DDITV02 DD statement. The subsystem member name is specified by the parameter SSM= on the DL/I batch invocation procedure.

- Optionally specify an output data set using the DDOTV02 DD statement. You might need this data set to receive messages from the IMS attachment facility about indoubt threads and diagnostic information.

Program design considerations for using DL/I batch

Address spaces in DL/I batch:

A DL/I batch region is independent of both the IMS control region and the CICS address space. The DL/I batch region loads the DL/I code into the application region along with the application program.

Commits in DL/I batch:

Commit IMS batch applications frequently so that you do not use resources for an extended time.

SQL statements and IMS calls in DL/I batch:

DL/I batch applications cannot use the SQL COMMIT and ROLLBACK statements; otherwise, you get an SQL error code. DLI/I batch applications also cannot use ROLS, SETS, and SYNC calls; otherwise the application program abnormally terminates.

Checkpoint calls in DL/I batch:

Write your program with SQL statements and DL/I calls, and use checkpoint calls. The frequency of checkpoints depends on the application design. All checkpoints that are issued by a batch application program must be unique. At a checkpoint, DL/I positioning is lost, DB2 cursors are closed (with the possible exception of cursors that are defined as WITH HOLD), commit duration locks are freed (again with some exceptions), and database changes are considered permanent to both IMS and DB2.

Application program synchronization in DL/I batch:

You can design an application program without using IMS checkpoints. In that case, if the program abnormally terminates before completing, DB2 backs out any updates, and you can use the IMS batch backout utility to back out the DL/I changes.

You can also have IMS dynamically back out the updates within the same job. You must specify the BKO parameter as 'Y' and allocate the IMS log to DASD.

You could have a problem if the system on which the job is run fails after the program terminates but before the job step ends. If you do not have a checkpoint call before the program ends, DB2 commits the unit of work without involving IMS. If the system fails before DL/I commits the data, the DB2 data is out of synchronization with the DL/I changes. If the system fails during DB2 commit processing, the DB2 data could be indoubt. When you restart the application program, use the XRST call to obtain checkpoint information and resolve any DB2 indoubt work units.

Recommendation: Always issue a symbolic checkpoint at the end of any update job to coordinate the commit of the outstanding unit of work for IMS and DB2.

Checkpoint and XRST considerations in DL/I batch:

If you use an XRST call, DB2 assumes that any checkpoint that is issued is a symbolic checkpoint. The options of the symbolic checkpoint call differ from the options of a basic checkpoint call. Using the incorrect form of the checkpoint call can cause problems.

If you do not use an XRST call, DB2 assumes that any checkpoint call that is issued is a basic checkpoint.

To make restart easier, use EBCDIC characters for checkpoint IDs.

When an application program needs to be restartable, you must use symbolic checkpoint and XRST calls. If you use an XRST call, it must be the first IMS call that is issued, and it must occur before any SQL statement. Also, you must use only one XRST call.

Synchronization call abends in DL/I batch:

If the application program contains an incorrect IMS synchronization call (CHKP, ROLB, ROLL, or XRST), causing IMS to issue a bad status code in

the PCB, DB2 abends the application program. Be sure to test these calls before placing the programs in production.

Related concepts:

“Input and output data sets for DL/I batch jobs” on page 975

 Multiple system consistency (DB2 Administration Guide)

Related tasks:

Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915

Chapter 13. Invoking a user-defined function

You can use a user-defined function wherever you can use a built-in function.

Before you invoke a user-defined function, review the following:

- How DB2 resolves functions
- Cases when DB2 casts arguments for a user-defined function
- Abnormal termination of an external user-defined function
- Syntax for invoking a table function, which is in the from-clause
- Syntax for invoking a user-defined scalar function, which is explained in function invocation

You can invoke a sourced or external user-defined scalar function in an SQL statement wherever you use an expression. For a table function, you can invoke the user-defined function only in the FROM clause of a SELECT statement. The invoking SQL statement can be in a stand alone program, a stored procedure, a trigger body, or another user-defined function.

Recommendations for invoking user-defined functions:

Invoke user-defined functions with external actions and nondeterministic user-defined functions from select lists: Invoking user-defined functions with external action from a select list and nondeterministic user-defined functions from a select list is preferred to invoking these user-defined functions from a predicate.

The access path that DB2 chooses for a predicate determines whether a user-defined function in that predicate is executed. To ensure that DB2 executes the external action for each row of the result table, put the user-defined function invocation in the SELECT list.

Invoking a nondeterministic user-defined function from a predicate can yield undesirable results. The following example demonstrates this idea.

Suppose that you execute this query:

```
SELECT COUNTER(), C1, C2 FROM T1 WHERE COUNTER() = 2;
```

Table T1 looks like this:

C1	C2
1	b
2	c
3	a

COUNTER is a user-defined function that increments a variable in the scratchpad each time it is invoked.

DB2 invokes an instance of COUNTER in the predicate 3 times. Assume that COUNTER is invoked for row 1 first, for row 2 second, and for row 3 third. Then COUNTER returns 1 for row 1, 2 for row 2, and 3 for row 3. Therefore, row 2 satisfies the predicate WHERE COUNTER()=2, so DB2 evaluates the SELECT list for row 2. DB2 uses a different instance of COUNTER in the select list from the

instance in the predicate. Because the instance of COUNTER in the select list is invoked only once, it returns a value of 1. Therefore, the result of the query is:

```
COUNTER() C1 C2
----- -- --
1         2 c
```

This is not the result you might expect.

The results can differ even more, depending on the order in which DB2 retrieves the rows from the table. Suppose that an ascending index is defined on column C2. Then DB2 retrieves row 3 first, row 1 second, and row 2 third. This means that row 1 satisfies the predicate WHERE COUNTER()=2. The value of COUNTER in the select list is again 1, so the result of the query in this case is:

```
COUNTER() C1 C2
----- -- --
1         1 b
```

Understand the interaction between scrollable cursors and nondeterministic user-defined functions or user-defined functions with external actions: When you use a scrollable cursor, you might retrieve the same row multiple times while the cursor is open. If the select list of the cursor's SELECT statement contains a user-defined function, that user-defined function is executed each time you retrieve a row. Therefore, if the user-defined function has an external action, and you retrieve the same row multiple times, the external action is executed multiple times for that row.

A similar situation occurs with scrollable cursors and nondeterministic functions. The result of a nondeterministic user-defined function can be different each time you execute the user-defined function. If the select list of a scrollable cursor contains a nondeterministic user-defined function, and you use that cursor to retrieve the same row multiple times, the results can differ each time you retrieve the row.

A nondeterministic user-defined function in the predicate of a scrollable cursor's SELECT statement does not change the result of the predicate while the cursor is open. DB2 evaluates a user-defined function in the predicate only once while the cursor is open.

Related concepts:

- “Abnormal termination of an external user-defined function” on page 554
- “Cases when DB2 casts arguments for a user-defined function” on page 788
- “How DB2 resolves functions” on page 780

 [Function invocation \(DB2 SQL\)](#)

Related reference:

 [from-clause \(DB2 SQL\)](#)

Determining the authorization ID for invoking user-defined functions

The authorization ID under which a user-defined function is invoked depends on whether the function was invoked statically or dynamically.

If your user-defined function is invoked:	The authorization ID under which the user-defined function is invoked is:
statically	The owner of the package that contains the user-defined function invocation.
dynamically	Dependent upon the value of bind parameter DYNAMICRULES for the package that contains the function invocation.

If the SQL statements in the user-defined function package execute:	The authorization ID is:
statically	The owner of the user-defined function package
dynamically	dependent upon the value of DYNAMICRULES with which the user-defined function package was bound.

The DYNAMICRULES bind parameter influences a number of characteristics of an application program.

Related concepts:

“DYNAMICRULES bind option” on page 959

Ensuring that DB2 executes the intended user-defined function

Multiple functions with the same name can exist in the same schema or in different schemas. You should take certain actions to ensure that DB2 chooses the correct function to execute.

When you use the following techniques, you can simplify function resolution:

- When you invoke a function, use the qualified name. This causes DB2 to search for functions only in the schema you specify. This has two advantages:
 - DB2 is less likely to choose a function that you did not intend to use. Several functions might fit the invocation equally well. DB2 picks the function whose schema name is earliest in the SQL path, which might not be the function you want.
 - The number of candidate functions is smaller, so DB2 takes less time for function resolution.
- Cast parameters in a user-defined function invocation to the types in the user-defined function definition. For example, if an input parameter for user-defined function FUNC is defined as DECIMAL(13,2), and the value you want to pass to the user-defined function is an integer value, cast the integer value to DECIMAL(13,2):

```
SELECT FUNC(CAST (INTCOL AS DECIMAL(13,2))) FROM T1;
```
- Use the data type BIGINT for numeric parameters in a user-defined function. If you use BIGINT as the parameter type, when you invoke the function, you can pass in SMALLINT, INTEGER, or BIGINT values. If you use SMALLINT or REAL as the parameter type, you must pass parameters of the same types. For

example, if user-defined function FUNC is defined with a parameter of type SMALLINT, only an invocation with a parameter of type SMALLINT resolves correctly. The following call does not resolve to FUNC because the constant 123 is of type INTEGER, not SMALLINT:

```
SELECT FUNC(123) FROM T1;
```

- Avoid defining user-defined function string parameters with fixed-length string types. If you define a parameter with a fixed-length string type (CHAR, GRAPHIC, or BINARY), you can invoke the user-defined function only with a fixed-length string parameter. However, if you define the parameter with a varying-length string type (VARCHAR, VARGRAPHIC, or VARBINARY), you can invoke the user-defined function with either a fixed-length string parameter or a varying-length string parameter.

If you must define parameters for a user-defined function as CHAR or BINARY, and you call the user-defined function from a C program or SQL procedure, you need to cast the corresponding parameter values in the user-defined function invocation to CHAR or BINARY to ensure that DB2 invokes the correct function. For example, suppose that a C program calls user-defined function CVRTNUM, which takes one input parameter of type CHAR(6). Also suppose that you declare host variable empnumbr as char empnumbr[6]. When you invoke CVRTNUM, cast empnumbr to CHAR:

```
UPDATE EMP
SET EMPNO=CVRTNUM(CHAR(:empnumbr))
WHERE EMPNO = :empnumbr;
```

How DB2 resolves functions

Function resolution is the process by which DB2 determines which user-defined function or built-in function to execute. You need to understand the function resolution process that DB2 uses to ensure that you invoke the user-defined function that you want to invoke.

Several user-defined functions with the same name but different numbers or types of parameters can exist in a DB2 subsystem. Several user-defined functions with the same name can have the same number of parameters, as long as the data types of any of the first 30 parameters are different. In addition, several user-defined functions might have the same name as a built-in function. When you invoke a function, DB2 must determine which user-defined function or built-in function to execute.

DB2 performs these steps for function resolution:

1. Determines if any function instances are candidates for execution. If no candidates exist, DB2 issues an SQL error message.
2. Compares the data types of the input parameters to determine which candidates fit the invocation best.

DB2 does not compare data types for input parameters that are untyped parameter markers.

For a qualified function invocation, if there are no parameter markers in the invocation, the result of the data type comparison is one best fit. That best fit is the choice for execution. If there are parameter markers in the invocation, there might be more than one best fit. DB2 issues an error if there is more than one best fit.

For an unqualified function invocation, DB2 might find multiple best fits because the same function name with the same input parameters can exist in different schemas, or because there are parameter markers in the invocation.

3. If two or more candidates fit the unqualified function invocation equally well because the same function name with the same input parameters exists in different schemas, DB2 chooses the user-defined function whose schema name is earliest in the SQL path.

For example, suppose functions SCHEMA1.X and SCHEMA2.X fit a function invocation equally well. Assume that the SQL path is:

"SCHEMA2", "SYSPROC", "SYSIBM", "SCHEMA1", "SYSFUN"

Then DB2 chooses function SCHEMA2.X.

If two or more candidates fit the unqualified function invocation equally well because the function invocation contains parameter markers, DB2 issues an error.

The remainder of this section discusses details of the function resolution process and gives suggestions on how you can ensure that DB2 picks the right function.

How DB2 chooses candidate functions:

An instance of a user-defined function is a candidate for execution only if it meets all of the following criteria:

- If the function name is qualified in the invocation, the schema of the function instance matches the schema in the function invocation.
If the function name is unqualified in the invocation, the schema of the function instance matches a schema in the invoker's SQL path.
- 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 input parameters in the function invocation.
- The function invoker is authorized to execute the function instance.
- The type of each of the input parameters in the function invocation matches or is *promotable* to the type of the corresponding parameter in the function instance.

If an input parameter in the function invocation is an untyped parameter marker, DB2 considers that parameter to be a match or promotable.

For a function invocation that passes a transition table, the data type, length, precision, and scale of each column in the transition table must match exactly the data type, length, precision, and scale of each column of the table that is named in the function instance definition. For information about transition tables, see "Creating triggers" on page 493.

- The create timestamp for a user-defined function must be older than the BIND or REBIND timestamp for the package or plan in which the user-defined function is invoked.

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

If you use an access control authorization exit routine, some user-defined functions that were not candidates for execution before the original BIND or REBIND of the invoking plan or package might become candidates for execution during the automatic rebind of the invoking plan or package.

If a user-defined function is invoked during an automatic rebind, and that user-defined function is invoked from a trigger body and receives a transition table, then the form of the invoked function that DB2 uses for function selection

includes only the columns of the transition table that existed at the time of the original BIND or REBIND of the package or plan for the invoking program.

During an automatic rebind, DB2 does not consider built-in functions for function resolution if those built-in functions were introduced in a later release of DB2 than the release in which the BIND or REBIND of the invoking plan or package occurred.

When you explicitly bind or rebind a plan or package, the plan or package receives a release dependency marker. When DB2 performs an automatic rebind of a query that contains a function invocation, a built-in function is a candidate for function resolution only if the release dependency marker of the built-in function is the same as or lower than the release dependency marker of the plan or package that contains the function invocation.

Example: Suppose that in this statement, the data type of A is SMALLINT:
SELECT USER1.ADDTWO(A) FROM TABLEA;

Two instances of USER1.ADDTWO are defined: one with an input parameter of type INTEGER and one with an input parameter of type DECIMAL. Both function instances are candidates for execution because the SMALLINT type is promotable to either INTEGER or DECIMAL. However, the instance with the INTEGER type is a better fit because INTEGER is higher in the list than DECIMAL.

How DB2 chooses the best fit among candidate functions:

More than one function instance might be a candidate for execution. In that case, DB2 determines which function instances are the best fit for the invocation by comparing parameter data types.

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

1. DB2 compares the data types of the first parameter in the function invocation to the data type of the first parameter in each function instance.
If the first parameter in the invocation is an untyped parameter marker, DB2 does not do the comparison.
2. For the first parameter, if one function instance has a data type that fits the function invocation better than the data types in the other instances, that function is a best fit.
3. If the data types of the first parameter are the same for all function instances, or if the first parameter in the function invocation is an untyped parameter marker, DB2 repeats this process for the next parameter. DB2 continues this process for each parameter until it finds a best fit.

Example of function resolution: Suppose that a program contains the following statement:

```
SELECT FUNC(VCHARCOL,SMINTCOL,DECCOL) FROM T1;
```

In user-defined function FUNC, VCHARCOL has data type VARCHAR, SMINTCOL has data type SMALLINT, and DECCOL has data type DECIMAL. Also suppose that two function instances with the following definitions meet the appropriate criteria and are therefore candidates for execution.


```
Candidate 1:
CREATE FUNCTION FUNC(VARCHAR(20),INTEGER,DOUBLE)
  RETURNS DECIMAL(9,2)
  EXTERNAL NAME 'FUNC1'
  PARAMETER STYLE SQL
  LANGUAGE COBOL;
```

```
Candidate 2:
CREATE FUNCTION FUNC(VARCHAR(20),REAL,DOUBLE)
  RETURNS DECIMAL(9,2)
  EXTERNAL NAME 'FUNC2'
  PARAMETER STYLE SQL
  LANGUAGE COBOL;
```

DB2 compares the data type of the first parameter in the user-defined function invocation to the data types of the first parameters in the candidate functions. Because the first parameter in the invocation has data type VARCHAR, and both candidate functions also have data type VARCHAR, DB2 cannot determine the better candidate based on the first parameter. Therefore, DB2 compares the data types of the second parameters.

The data type of the second parameter in the invocation is SMALLINT. INTEGER, which is the data type of candidate 1, is a better fit to SMALLINT than REAL, which is the data type of candidate 2. Therefore, candidate 1 is the DB2 choice for execution.

Related concepts:

 Promotion of data types (DB2 SQL)

Related tasks:

“Creating triggers” on page 493

Related information:

 Exit routines (DB2 Administration Guide)

Checking how DB2 resolves functions by using DSN_FUNCTION_TABLE

Because multiple user-defined functions can have the same name, you should ensure that DB2 invokes the function that you intended to invoke. One way to check that the correct function was invoked is to use a function table called DSN_FUNCTION_TABLE.


To check how DB2 resolves a function by using DSN_FUNCTION_TABLE:

1. If *your_userID*.DSN_FUNCTION_TABLE does not already exist, create this table by following the instructions in DSN_FUNCTION_TABLE (DB2 Performance).
2. Populate *your_userID*.DSN_FUNCTION_TABLE with information about which functions are invoked by a particular SQL statement by performing one of the following actions:
 - Execute the EXPLAIN statement on the SQL statement.
 - Ensure that the program that contains the SQL statement is bound with EXPLAIN(YES) and run the program.

DB2 puts a row in *your_userID*.DSN_FUNCTION_TABLE for each function that is referenced in each SQL statement.

3. Check the rows that were added to *your_userID.DSN_FUNCTION_TABLE* to ensure that the appropriate function was invoked. Use the following columns to help you find applicable rows: QUERYNO, APPLNAME, PROGNAME, COLLID, and EXPLAIN_TIME.

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

 EXPLAIN (DB2 SQL)

DSN_FUNCTION_TABLE

The function table, DSN_FUNCTION_TABLE, contains descriptions of functions that are used in specified SQL statements.

 PSPI

Recommendation: Do not manually insert data into system-maintained EXPLAIN tables, and use care when deleting obsolete EXPLAIN table data. The data is intended to be manipulated only by the DB2 EXPLAIN function and optimization tools. Certain optimization tools depend on instances of the various EXPLAIN tables. Be careful not to delete data from or drop instances EXPLAIN tables that are created for these tools.

Important: If the EXPLAIN tables have any format older than the Version 8 format, or are encoded in EBCDIC, DB2 returns an error for any operation that tries inserts rows in the EXPLAIN tables.

Qualifiers

Your subsystem or data sharing group can contain more than one of these tables:

SYSIBM

One instance of this table can be created with the SYSIBM qualifier. DB2 and SQL optimization tools might use the table and the data that it contains. The table is created when you run job DSNTIJSJ when you install or migrate DB2.

userID You can create additional instances of EXPLAIN tables that are qualified by user ID. These tables are populated with statement cost information when you issue the EXPLAIN statement or bind. They are also populated when you specify EXPLAIN(YES) or EXPLAIN(ONLY) in a BIND or REBIND command. SQL optimization tools might also create EXPLAIN tables that are qualified by a user ID. You can find the SQL statement for creating an instance of these tables in member DSNTESC of the SDSNSAMP library.

Sample CREATE TABLE statement

You can find a sample CREATE TABLE statement for each EXPLAIN table in member DSNTESC of the SDSNSAMP library.

Column descriptions

 PSPI

The following table describes the columns of DSN_FUNCTION_TABLE.

Table 129. Descriptions of columns in DSN_FUNCTION_TABLE

Column name	Data type	Description
QUERYNO	INTEGER NOT NULL WITH DEFAULT	<p>A number that identifies the statement that is being explained. The origin of the value depends on the context of the row:</p> <p>For rows produced by EXPLAIN statements The number specified in the QUERYNO clause, which is an optional part of the SELECT, INSERT, UPDATE, MERGE, and DELETE statement syntax.</p> <p>For rows not produced by EXPLAIN statements DB2 assigns a number that is based on the line number of the SQL statement in the source program.</p> <p>When the values of QUERYNO are based on the statement number in the source program, values that exceed 32767 are reported as 0. However, in certain rare cases, the value is not guaranteed to be unique.</p> <p>When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, if the QUERYNO clause is specified, then its value is used by DB2. Otherwise DB2 assigns a number based on the line number of the SQL statement in the non-inline SQL function, native SQL procedure.</p>
QBLOCKNO	INTEGER NOT NULL WITH DEFAULT	A number that identifies each query block within a query. The value of the numbers are not in any particular order, nor are they necessarily consecutive.
APPLNAME	VARCHAR(24) NOT NULL WITH DEFAULT	<p>The name of the application plan for the row. Applies only to embedded EXPLAIN statements that are executed from a plan or to statements that are explained when binding a plan. A blank indicates that the column is not applicable.</p> <p>When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.</p>
PROGNAME	VARCHAR(128) NOT NULL WITH DEFAULT	<p>The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. A blank indicates that the column is not applicable.</p> <p>When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.</p>
COLLID	VARCHAR(128) NOT NULL WITH DEFAULT	<p>The collection ID:</p> <p>DSNDYNAMICSQLCACHE The row originates from the dynamic statement cache</p> <p>DSNEXPLAINMODEYES The row originates from an application that specifies YES for the value of the CURRENT EXPLAIN MODE special register.</p> <p>DSNEXPLAINMODEEXPLAIN The row originates from an application that specifies EXPLAIN for the value of the CURRENT EXPLAIN MODE special register.</p> <p>When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.</p>

Table 129. Descriptions of columns in DSN_FUNCTION_TABLE (continued)

Column name	Data type	Description
GROUP_MEMBER	VARCHAR(24) NOT NULL WITH DEFAULT	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
EXPLAIN_TIME	TIMESTAMP NOT NULL WITH DEFAULT	The time when the EXPLAIN information was captured: All cached statements When the statement entered the cache, in the form of a full-precision timestamp value. Non-cached static statements When the statement was bound, in the form of a full precision timestamp value. Non-cached dynamic statements When EXPLAIN was executed, in the form of a value equivalent to a CHAR(16) representation of the time appended by 4 zeros.
SCHEMA_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	The schema name of the function invoked in the explained statement.
FUNCTION_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	The name of the function invoked in the explained statement.
SPEC_FUNC_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	The specific name of the function invoked in the explained statement.
FUNCTION_TYPE	CHAR(2) NOT NULL WITH DEFAULT	The type of function invoked in the explained statement. Possible values are: CU Column function SU Scalar function TU Table function
VIEW_CREATOR	VARCHAR(128) NOT NULL WITH DEFAULT	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the creator of the view. Otherwise, blank.
VIEW_NAME	VARCHAR(128) NOT NULL WITH DEFAULT	If the function specified in the FUNCTION_NAME column is referenced in a view definition, the name of the view. Otherwise, blank.
PATH	VARCHAR(2048) NOT NULL WITH DEFAULT	The value of the SQL path that was used to resolve the schema name of the function.
FUNCTION_TEXT	VARCHAR(1500) NOT NULL WITH DEFAULT	The text of the function reference (the function name and parameters). If the function reference is over 100 bytes, this column contains the first 100 bytes. For functions specified in infix notation, FUNCTION_TEXT contains only the function name. For example, for a function named /, which overloads the SQL divide operator, if the function reference is A/B, FUNCTION_TEXT contains only /.
FUNC_VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	For a version of a non-inline SQL scalar function, this column contains the version identifier. For all other cases, this column contains a zero length string. A version of a non-inline SQL scalar function is defined in the SYSIBM.SYSROUTINES table with ORIGIN='Q', FUNCTION_TYPE='S', INLINE='N', and VERSION column containing the version identifier.

Table 129. Descriptions of columns in DSN_FUNCTION_TABLE (continued)

Column name	Data type	Description
SECURE	CHAR(1) NOT NULL WITH DEFAULT	Whether the user-defined function is secure.
SECTNOI	INTEGER NOT NULL WITH DEFAULT	The section number of the statement. The value is taken from the same column in SYSPACKSTMT or SYSSTMT tables and can be used to join tables to reconstruct the access path for the statement. This column is applicable only for static statements. The default value of -1 indicates EXPLAIN information that was captured in Version 9 or earlier.
VERSION	VARCHAR(122) NOT NULL WITH DEFAULT	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. A blank indicates that the column is not applicable. When the SQL statement is embedded in a non-inline SQL function or native SQL procedure, this column is not used and is blank.



Related tasks:

- ➡ Checking how DB2 resolves functions by using DSN_FUNCTION_TABLE (DB2 Application programming and SQL)

Restrictions when passing arguments with distinct types to functions

Because DB2 enforces strong typing when you pass arguments to a function, you must follow certain rules when passing arguments with distinct types to functions.

Adhere to the following rules:

- You can pass arguments that have distinct types to a function if either of the following conditions is true:
 - A version of the function that accepts those distinct types is defined.
This also applies to infix operators. If you want to use one of the five built-in infix operators (||, /, *, +, -) with your distinct types, you must define a version of that operator that accepts the distinct types.
 - You can cast your distinct types to the argument types of the function.
- If you pass arguments to a function that accepts only distinct types, the arguments you pass must have the same distinct types as in the function definition. If the types are different, you must cast your arguments to the distinct types in the function definition.
If you pass constants or host variables to a function that accepts only distinct types, you must cast the constants or host variables to the distinct types that the function accepts.

The following examples demonstrate how to use distinct types as arguments in function invocations.

Example: Defining a function with distinct types as arguments: Suppose that you want to invoke the built-in function HOUR with a distinct type that is defined like this:

```
CREATE DISTINCT TYPE FLIGHT_TIME AS TIME;
```

The HOUR function takes only the TIME or TIMESTAMP data type as an argument, so you need a sourced function that is based on the HOUR function that accepts the FLIGHT_TIME data type. You might declare a function like this:

```
CREATE FUNCTION HOUR(FLIGHT_TIME)
  RETURNS INTEGER
  SOURCE SYSIBM.HOUR(TIME);
```

Example: Casting function arguments to acceptable types: Another way you can invoke the HOUR function is to cast the argument of type FLIGHT_TIME to the TIME data type before you invoke the HOUR function. Suppose table FLIGHT_INFO contains column DEPARTURE_TIME, which has data type FLIGHT_TIME, and you want to use the HOUR function to extract the hour of departure from the departure time. You can cast DEPARTURE_TIME to the TIME data type, and then invoke the HOUR function:

```
SELECT HOUR(CAST(DEPARTURE_TIME AS TIME)) FROM FLIGHT_INFO;
```

Example: Using an infix operator with distinct type arguments: Suppose you want to add two values of type US_DOLLAR. Before you can do this, you must define a version of the + function that accepts values of type US_DOLLAR as operands:

```
CREATE FUNCTION "+"(US_DOLLAR,US_DOLLAR)
  RETURNS US_DOLLAR
  SOURCE SYSIBM."+"(DECIMAL(9,2),DECIMAL(9,2));
```

Because the US_DOLLAR type is based on the DECIMAL(9,2) type, the source function must be the version of + with arguments of type DECIMAL(9,2).

Example: Casting constants and host variables to distinct types to invoke a user-defined function: Suppose function CDN_TO_US is defined like this:

```
CREATE FUNCTION EURO_TO_US(EURO)
  RETURNS US_DOLLAR
  EXTERNAL NAME 'CDNCVT'
  PARAMETER STYLE SQL
  LANGUAGE C;
```

This means that EURO_TO_US accepts only the EURO type as input. Therefore, if you want to call CDN_TO_US with a constant or host variable argument, you must cast that argument to distinct type EURO:

```
SELECT * FROM US_SALES
  WHERE TOTAL = EURO_TO_US(EURO(:H1));
SELECT * FROM US_SALES
  WHERE TOTAL = EURO_TO_US(EURO(10000));
```

Cases when DB2 casts arguments for a user-defined function

In certain situations, when you invoke a user-defined function, DB2 casts your input argument values to different data types and lengths.

Whenever you invoke a user-defined function, DB2 assigns your input argument values to parameters with the data types and lengths in the user-defined function definition.

When you invoke a user-defined function that is sourced on another function, DB2 casts your arguments to the data types and lengths of the sourced function.

The following example demonstrates what happens when the parameter definitions of a sourced function differ from those of the function on which it is sourced.

Suppose that external user-defined function TAXFN1 is defined like this:

```
CREATE FUNCTION TAXFN1(DEC(6,0))
  RETURNS DEC(5,2)
  PARAMETER STYLE SQL
  LANGUAGE C
  EXTERNAL NAME TAXPROG;
```

Sourced user-defined function TAXFN2, which is sourced on TAXFN1, is defined like this:

```
CREATE FUNCTION TAXFN2(DEC(8,2))
  RETURNS DEC(5,0)
  SOURCE TAXFN1;
```

You invoke TAXFN2 using this SQL statement:

```
UPDATE TB1
  SET SALESTAX2 = TAXFN2(PRICE2);
```

TB1 is defined like this:

```
CREATE TABLE TB1
  (PRICE1 DEC(6,0),
  SALESTAX1 DEC(5,2),
  PRICE2 DEC(9,2),
  SALESTAX2 DEC(7,2));
```

Now suppose that PRICE2 has the DECIMAL(9,2) value 0001234.56. DB2 must first assign this value to the data type of the input parameter in the definition of TAXFN2, which is DECIMAL(8,2). The input parameter value then becomes 001234.56. Next, DB2 casts the parameter value to a source function parameter, which is DECIMAL(6,0). The parameter value then becomes 001234. (When you cast a value, that value is truncated, rather than rounded.)

Now, if TAXFN1 returns the DECIMAL(5,2) value 123.45, DB2 casts the value to DECIMAL(5,0), which is the result type for TAXFN2, and the value becomes 00123. This is the value that DB2 assigns to column SALESTAX2 in the UPDATE statement.

Casting of parameter markers

You can use untyped parameter markers in a function invocation. However, DB2 cannot compare the data types of untyped parameter markers to the data types of candidate functions. Therefore, DB2 might find more than one function that qualifies for invocation. If this happens, an SQL error occurs. To ensure that DB2 picks the right function to execute, cast the parameter markers in your function invocation to the data types of the parameters in the function that you want to execute. For example, suppose that two versions of function FX exist. One version of FX is defined with a parameter of type of DECIMAL(9,2), and the other is defined with a parameter of type INTEGER. You want to invoke FX with a parameter marker, and you want DB2 to execute the version of FX that has a DECIMAL(9,2) parameter. You need to cast the parameter marker to a DECIMAL(9,2) type by using a CAST specification:

```
SELECT FX(CAST(? AS DECIMAL(9,2))) FROM T1;
```

Related concepts:

 Assignment and comparison (DB2 SQL)

Chapter 14. Calling a stored procedure from your application

To run a stored procedure, you can either call it from a client program or invoke it from the command line processor.

Before you call a stored procedure, ensure that you have all of the following authorizations that are required to run the stored procedure:

- Authorization to execute the stored procedure that is referenced in the CALL statement.

The authorizations that you need depend on whether the form of the CALL statement is *CALL procedure-name* or *CALL :host-variable*.

- Authorization to execute any triggers or user-defined functions that the stored procedure invokes.
- Authorization to execute the stored procedure package and any packages under the stored procedure package.

For example, if the stored procedure invokes any user-defined functions, you need authorization to execute the packages for those user-defined functions.

An application program that calls a stored procedure can perform one or more of the following actions:

- Call more than one stored procedure.
- Call a single stored procedure more than once at the same or at different levels of nesting. However, do not assume that the variables for the stored procedures persist between calls.

If a stored procedure runs as a main program, before each call, Language Environment reinitializes the storage that is used by the stored procedure. Program variables for the stored procedure do not persist between calls.

If a stored procedure runs as a subprogram, Language Environment does not initialize the storage between calls. Program variables for the stored procedure can persist between calls. However, you should not assume that your program variables are available from one stored procedure call to another call for the following reasons:

- Stored procedures from other users can run in an instance of Language Environment between two executions of your stored procedure.
- Consecutive executions of a stored procedure might run in different stored procedure address spaces.
- The z/OS operator might refresh Language Environment between two executions of your stored procedure.
- Call a local or remote stored procedure.

If both the client and server application environments support two-phase commit, the coordinator controls updates between the application, the server, and the stored procedures. If either side does not support two-phase commit, updates fail.
- Mix CALL statements with other SQL statements.
- Use any of the DB2 attachment facilities.

DB2 runs stored procedures under the DB2 thread of the calling application, which means that the stored procedures are part of the caller's unit of work.

| **JDBC and ODBC applications:** These instructions do not apply to JDBC and
| ODBC applications. Instead, see the following information for how to call stored
| procedures from those applications:

- | • For ODBC applications, see *Stored procedure calls in a DB2 ODBC application*
| (DB2 Programming for ODBC).
- | • For JDBC applications, see *Calling stored procedures in JDBC applications* (DB2
| Application Programming for Java)

| To call a stored procedure from your application:

- | 1. Assign values to the IN and INOUT parameters.
- | 2. Optional: To improve application performance, initialize the length of LOB
| output parameters to zero.
- | 3. If the stored procedure exists at a remote location, perform the following
| actions:

- a. Assign values to the OUT parameters.

| When you call a stored procedure at a remote location, the local DB2
| server cannot determine whether the parameters are input (IN) or output
| (OUT or INOUT) parameters. Therefore, you must initialize the values of
| all output parameters before you call a stored procedure at a remote
| location.

- b. Optional: Issue an explicit CONNECT statement to connect to the remote
| server.

| If you do not issue this statement explicitly, you can implicitly connect to
| the server by using a three-part name to identify the stored procedure in
| the next step.

| The advantage of issuing an explicit CONNECT statement is that your
| CALL statement, which is described in the next step, is portable to other
| operating systems. The advantage of implicitly connecting is that you do
| not need to issue this extra CONNECT statement.

| **Requirement:** When deciding whether to implicitly or explicitly connect to
| the remote server, consider the requirement for programs that execute the
| ASSOCIATE LOCATORS or DESCRIBE PROCEDURE statements. You
| must use the same form of the procedure name on the CALL statement
| and on the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE
| statement.

4. Invoke the stored procedure with the SQL CALL statement. Make sure that
| you pass parameter data types that are compatible.

| If the stored procedure exists on a remote server and you did not issue an
| explicit CONNECT statement, specify a three-part name to identify the stored
| procedure, and implicitly connect to the server where the stored procedure is
| located.

| For native SQL procedures, the active version of the stored procedure is
| invoked by default. Optionally, you can specify a version of the stored
| procedure other than the active version.

| To allow null values for parameters, use indicator variables.

5. Optional: Retrieve the status of the procedure.
6. Process any output, including the OUT and INOUT parameters.
7. If the stored procedure returns multiple result sets, retrieve those result sets.

Recommendation: Close the result sets after you retrieve them, and issue frequent commits to prevent DB2 storage shortages and EDM POOL FULL conditions.

8. For PL/I applications, also perform the following actions:
 - a. Include the run time option NOEXECOPS in your source code.
 - b. Specify the compile-time option SYSTEM(MVS).

These additional steps ensure that the linkage conventions work correctly on z/OS.

9. For C applications, include the following line in your source code:

```
#pragma runopts(PLIST(OS))
```

This code ensures that the linkage conventions work correctly on z/OS.

This option is not applicable to other operating systems. If you plan to use a C stored procedure on other platforms besides z/OS, use one of the forms of conditional compilation, as shown in the following example, to include this option only when you compile on z/OS.

Form 1:

```
#ifndef MVS  
#pragma runopts(PLIST(OS))  
#endif
```

Form 2:

```
#ifndef WKSTN  
#pragma runopts(PLIST(OS))  
#endif
```

10. Prepare the application as you would any other application by precompiling, compiling, and link-editing the application and binding the DBRM.

If the application calls a remote stored procedure, perform the following additional steps when you bind the DBRM:

- Bind the DBRM into a package at the local DB2 server. Use the bind option DBPROTOCOL(DRDA). If the stored procedure name cannot be resolved until run time, also specify the bind option VALIDATE(RUN). The stored procedure name might not be resolved at run time if you use a variable for the stored procedure name or if the stored procedure exists on a remote server.
- Bind the DBRM into a package at the remote DB2 server. If your client program accesses multiple servers, bind the program at each server.
- Bind all packages into a plan at the local DB2 server. Use the bind option DBPROTOCOL(DRDA).

11. Ensure that stored procedure completed successfully.

If a stored procedure abnormally terminates, DB2 performs the following actions:

- The calling program receives an SQL error as notification that the stored procedure failed.
- DB2 places the calling program's unit of work in a must-rollback state.
- DB2 stops the stored procedure, and subsequent calls fail, in either of the following conditions:
 - The number of abnormal terminations equals the STOP AFTER *n* FAILURES value for the stored procedure.
 - The number of abnormal terminations equals the default MAX ABEND COUNT value for the subsystem.

- The stored procedure does not handle the abend condition, and DB2 refreshes the environment for Language Environment to recover the storage that the application uses. In most cases, the environment does not need to restart.
- A data set is allocated in the DD statement CEEDUMP in the JCL procedure that starts the stored procedures address space. In this case, Language Environment writes a small diagnostic dump to this data set. Use the information in the dump to debug the stored procedure.
- In a data sharing environment, the stored procedure is placed in STOPABN status only on the member where the abends occurred. A calling program can invoke the stored procedure from other members of the data sharing group. The status on all other members is STARTED.

Example of simple CALL statement: The following example shows a simple CALL statement that you might use to invoke stored procedure A:

```
EXEC SQL CALL A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, :CODE);
```

In this example, :EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, and :CODE are host variables that you have declared earlier in your application program.

Example of using a host structure for multiple parameter values: Instead of passing each parameter separately, as shown in the example of a simple CALL statement, you can pass them together as a host structure. For example, assume that you defined the following host structure in your application:

```
struct {
    char EMP[7];
    char PRJ[7];
    short ACT;
    short EMT;
    char EMS[11];
    char EME[11];
} empstruc;
```

You can then issue the following CALL statement to invoke stored procedure A:

```
EXEC SQL CALL A (:empstruc, :TYPE, :CODE);
```

Examples of calling a remote stored procedure: Suppose that stored procedure A is in schema SCHEMAA at remote location LOCA. To invoke stored procedure A, you can explicitly or implicitly connect to the server:

- The following example shows how to explicitly connect to LOCA and then issue a CALL statement:

```
EXEC SQL CONNECT TO LOCA;
EXEC SQL CALL SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,
    :TYPE, :CODE);
```

- The following example shows how to implicitly connect to LOCA by specifying the three-part name for stored procedure A in the CALL statement:

```
EXEC SQL CALL LOCA.SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS,
    :EME, :TYPE, :CODE);
```

Example of passing parameters that can have null values: The preceding examples assume that none of the input parameters can have null values. The following example shows how to allow for null values for the parameters by passing indicator variables in the parameter list:

```
EXEC SQL CALL A (:EMP :IEMP, :PRJ :IPRJ, :ACT :IACT,
    :EMT :IEMT, :EMS :IEMS, :EME :IEME,
    :TYPE :ITYPE, :CODE :ICODE);
```

In this example, :IEMP, :IPRJ, :IACT, :IEMT, :IEMS, :IEME, :ITYPE, and :ICODE are indicator variables for the parameters.

Example of passing string constants and null values: The following example CALL statement passes integer and character string constants, a null value, and several host variables:

```
EXEC SQL CALL A ('000130', 'IF1000', 90, 1.0, NULL, '2009-10-01',  
                :TYPE, :CODE);
```

Example of using a host variable for the stored procedure name: The following example CALL statement uses a host variable for the name of the stored procedure:

```
EXEC SQL CALL :procnm (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,  
                      :TYPE, :CODE);
```

Assume that the stored procedure name is A. The host variable *procnm* is a character variable of length 255 or less that contains the value 'A'. Use this technique if you do not know in advance the name of the stored procedure, but you do know the parameter list convention.

Example of using an SQLDA to pass parameters in a single structure: The following example CALL statement shows how to pass parameters in a single structure, the SQLDA, rather than as separate host variables:

```
EXEC SQL CALL A USING DESCRIPTOR :sqlda;
```

sqlda is the name of an SQLDA.

One advantage of using an SQLDA is that you can change the encoding scheme of the stored procedure parameter values. For example, if the subsystem on which the stored procedure runs has an EBCDIC encoding scheme, and you want to retrieve data in ASCII CCSID 437, you can specify the CCSIDs for the output parameters in the SQLVAR fields of the SQLDA.

This technique for overriding the CCSIDs of parameters is the same as the technique for overriding the CCSIDs of variables. This technique involves including dynamic SQL for varying-list SELECT statements in your program. When you use this technique, the defined encoding scheme of the parameter must be different from the encoding scheme that you specify in the SQLDA. Otherwise, no conversion occurs.

The defined encoding scheme for the parameter is the encoding scheme that you specify in the CREATE PROCEDURE statement. If you do not specify an encoding scheme in this statement, the defined encoding scheme for the parameter is the default encoding scheme for the subsystem.

Example of a reusable CALL statement: Because the following example CALL statement uses a host variable name for the stored procedure and an SQLDA for the parameter list, it can be reused to call different stored procedures with different parameter lists:

```
EXEC SQL CALL :procnm USING DESCRIPTOR :sqlda;
```

Your client program must assign a stored procedure name to the host variable *procnm* and load the SQLDA with the parameter information before issuing the SQL CALL statement.

Related concepts:

“Stored procedure parameters” on page 561

Related tasks:

“Including dynamic SQL for varying-list SELECT statements in your program” on page 202

Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915

➤ Managing authorization for stored procedures (Managing Security)

➤ Temporarily overriding the active version of a native SQL procedure (DB2 Application programming and SQL)

Related reference:

➤ Statements (DB2 SQL)

➤ Sample scenarios of program preparations (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

Passing large output parameters to stored procedures by using indicator variables

If any output parameters occupy a large amount of storage, passing the entire storage area to a stored procedure can degrade performance. Instead, consider using indicator variables in the calling program to pass only a 2-byte area to the stored procedure and receive the entire area from the stored procedure.

You can use the following procedure regardless of whether the linkage convention for the stored procedure is GENERAL, GENERAL WITH NULLS, or SQL.

To pass large output parameters to stored procedures by using indicator variables:

1. Declare an indicator variable for every large output parameter in the stored procedure. If you are using the GENERAL WITH NULLS or SQL linkage convention, you must declare indicator variables for all of your parameters. In this case, you do not need to declare another indicator variable.
2. Assign a negative value to each indicator variable that is associated with a large output variable.
3. Include the indicator variables in the CALL statement.

For example, suppose that a stored procedure that is defined with the GENERAL linkage convention takes one integer input parameter and one character output parameter of length 6000. You do not want to pass the 6000 byte storage area to the stored procedure. The following example PL/I program passes only 2 bytes to the stored procedure for the output variable and receives all 6000 bytes from the stored procedure:

```
DCL INTVAR BIN FIXED(31);      /* This is the input variable */
DCL BIGVAR(6000);            /* This is the output variable */
DCL I1 BIN FIXED(15);        /* This is an indicator variable */
:
:
I1 = -1;                      /* Setting I1 to -1 causes only */
                              /* a two byte area representing */
                              /* I1 to be passed to the */
                              /* stored procedure, instead of */
                              /* the 6000 byte area for BIGVAR*/
EXEC SQL CALL PROCX(:INTVAR, :BIGVAR INDICATOR :I1);
```

Related reference:

“Linkage conventions for external stored procedures” on page 619

Data types for calling stored procedures

The data types that are available for calling applications are the same as the data types that are used when retrieving or updating stored procedures.

The format of the parameters that you pass in the CALL statement in an application must be compatible with the data types of the parameters in the CREATE PROCEDURE statement.

For languages other than REXX

For all data types except LOBs, ROWIDs, locators, and VARCHARs (for C language), see the tables listed in the following table for the host data types that are compatible with the data types in the stored procedure definition.

Table 130. Listing of tables of compatible data types

Language	Compatible data types table
Assembler	"Equivalent SQL and assembler data types" on page 261
C	"Equivalent SQL and C data types" on page 301
COBOL	"Equivalent SQL and COBOL data types" on page 352
PL/I	"Equivalent SQL and PL/I data types" on page 423

Calling a stored procedure from a REXX procedure

The format of the parameters that you pass in the CALL statement in a REXX procedure must be compatible with the data types of the parameters in the CREATE PROCEDURE statement.

The following table lists each SQL data type that you can specify for the parameters in the CREATE PROCEDURE statement and the corresponding format for a REXX parameter that represents that data type.

Table 131. Parameter formats for a CALL statement in a REXX procedure

SQL data type	REXX format
SMALLINT INTEGER BIGINT	A string of numerics that does not contain a decimal point or exponent identifier. The first character can be a plus or minus sign. This format also applies to indicator variables that are passed as parameters.
DECIMAL(<i>p,s</i>) NUMERIC(<i>p,s</i>)	A string of numerics that has a decimal point but no exponent identifier. The first character can be a plus or minus sign.
REAL FLOAT(<i>n</i>) DOUBLE DECFLOAT	A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus or minus sign and a series of numerics).
CHARACTER(<i>n</i>) VARCHAR(<i>n</i>) VARCHAR(<i>n</i>) FOR BIT DATA	A string of length <i>n</i> , enclosed in single quotation marks.
GRAPHIC(<i>n</i>) VARGRAPHIC(<i>n</i>)	The character G followed by a string enclosed in single quotation marks. The string within the quotation marks begins with a shift-out character (X'0E') and ends with a shift-in character (X'0F'). Between the shift-out character and shift-in character are <i>n</i> double-byte characters.

Table 131. Parameter formats for a CALL statement in a REXX procedure (continued)

SQL data type	REXX format
BINARY VARBINARY	Recommendation: Pass BINARY and VARBINARY values by using the SQLDA. If you specify an SQLDA when you call the stored procedure, set the SQLTYPE in the SQLDA. SQLDATA is a string of characters. If you use host variables, the REXX format of BINARY and VARBINARY data is BX followed by a string that is enclosed in a single quotation mark.
DATE	A string of length 10, enclosed in single quotation marks. The format of the string depends on the value of field DATE FORMAT that you specify when you install DB2.
TIME	A string of length 8, enclosed in single quotation marks. The format of the string depends on the value of field TIME FORMAT that you specify when you install DB2.
TIMESTAMP	A string of length 19 to 32, enclosed in single quotation marks. The string has the format <i>yyyy-mm-dd-hh.mm.ss</i> or <i>yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn</i> , where the number of fractional second digits can range from 0 to 12.
TIMESTAMP WITH TIME ZONE	A string of length 148 to 161, enclosed in single quotation marks. The string has the format <i>yyyymm- dd-hh.mm.ss.nnnnnnnnnnnn±th:tm</i> or <i>yyyymm- dd-hh.mm.ss.nnnnnnnnnnnn ±th:tm</i> , where the number of fractional second digits can range from 0 to 12
XML	No equivalent.

The following figure demonstrates how a REXX procedure calls the stored procedure in “REXX stored procedures” on page 650. The REXX procedure performs the following actions:

- Connects to the DB2 subsystem that was specified by the REXX procedure invoker.
- Calls the stored procedure to execute a DB2 command that was specified by the REXX procedure invoker.
- Retrieves rows from a result set that contains the command output messages.

```

/* REXX */
PARSE ARG SSID COMMAND /* Get the SSID to connect to */
/* and the DB2 command to be */
/* executed */
/*****/
/* Set up the host command environment for SQL calls. */
/*****/
"SUBCOM DSNREXX" /* Host cmd env available? */
IF RC THEN /* No--make one */
 S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
/*****/
/* Connect to the DB2 subsystem. */
/*****/
ADDRESS DSNREXX "CONNECT" SSID
IF SQLCODE ~= 0 THEN CALL SQLCA
PROC = 'COMMAND'
RESULTSIZ = 32703
RESULT = LEFT(' ',RESULTSIZ,' ')
/*****/
/* Call the stored procedure that executes the DB2 command. */
/* The input variable (COMMAND) contains the DB2 command. */
/* The output variable (RESULT) will contain the return area */
/* from the IFI COMMAND call after the stored procedure */
/* executes. */
/*****/

```



```

ADDRESS DSNREXX "EXECSQL" ,
"CALL" PROC "(:COMMAND, :RESULT)"
IF SQLCODE < 0 THEN CALL SQLCA
SAY 'RETCODE ='RETCODE
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD ='SQLERRD.1',',
      | SQLERRD.2',',
      | SQLERRD.3',',
      | SQLERRD.4',',
      | SQLERRD.5',',
      | SQLERRD.6
SAY 'SQLWARN ='SQLWARN.0',',
      | SQLWARN.1',',
      | SQLWARN.2',',
      | SQLWARN.3',',
      | SQLWARN.4',',
      | SQLWARN.5',',
      | SQLWARN.6',',
      | SQLWARN.7',',
      | SQLWARN.8',',
      | SQLWARN.9',',
      | SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
SAY C2X(RESULT) " "||RESULT||" "

/*****
/* Display the IFI return area in hexadecimal. */
*****/
OFFSET = 4+1
TOTLEN = LENGTH(RESULT)
DO WHILE ( OFFSET < TOTLEN )
  LEN = C2D(SUBSTR(RESULT,OFFSET,2))
  SAY SUBSTR(RESULT,OFFSET+4,LEN-4-1)
  OFFSET = OFFSET + LEN
END
/*****
/* Get information about result sets returned by the */
/* stored procedure. */
*****/
ADDRESS DSNREXX "EXECSQL DESCRIBE PROCEDURE :PROC INTO :SQLDA"
IF SQLCODE ^= 0 THEN CALL SQLCA
DO I = 1 TO SQLDA.SQLD
  SAY "SQLDA."I".SQLNAME ="SQLDA.I.SQLNAME";"
  SAY "SQLDA."I".SQLTYPE ="SQLDA.I.SQLTYPE";"
  SAY "SQLDA."I".SQLLOCATOR ="SQLDA.I.SQLLOCATOR";"
END I
/*****
/* Set up a cursor to retrieve the rows from the result */
/* set. */
*****/
ADDRESS DSNREXX "EXECSQL ASSOCIATE LOCATOR (:RESULT) WITH PROCEDURE :PROC"
IF SQLCODE ^= 0 THEN CALL SQLCA
SAY RESULT
ADDRESS DSNREXX "EXECSQL ALLOCATE C101 CURSOR FOR RESULT SET :RESULT"
IF SQLCODE ^= 0 THEN CALL SQLCA
CURSOR = 'C101'
ADDRESS DSNREXX "EXECSQL DESCRIBE CURSOR :CURSOR INTO :SQLDA"
IF SQLCODE ^= 0 THEN CALL SQLCA
/*****
/* Retrieve and display the rows from the result set, which */
/* contain the command output message text. */
*****/
DO UNTIL(SQLCODE ^= 0)
  ADDRESS DSNREXX "EXECSQL FETCH C101 INTO :SEQNO, :TEXT"
  IF SQLCODE = 0 THEN
    DO

```

```

        SAY TEXT
    END
END
IF SQLCODE = 0 THEN CALL SQLCA
ADDRESS DSNREXX "EXECSQL CLOSE C101"
IF SQLCODE = 0 THEN CALL SQLCA
ADDRESS DSNREXX "EXECSQL COMMIT"
IF SQLCODE = 0 THEN CALL SQLCA

/*****
/* Disconnect from the DB2 subsystem.
*****/
ADDRESS DSNREXX "DISCONNECT"
IF SQLCODE = 0 THEN CALL SQLCA
/*****
/* Delete the host command environment for SQL.
*****/
S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX') /* REMOVE CMD ENV */
RETURN
/*****
/* Routine to display the SQLCA
*****/
SQLCA:
TRACE 0
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD ='SQLERRD.1',',
        | SQLERRD.2',',
        | SQLERRD.3',',
        | SQLERRD.4',',
        | SQLERRD.5',',
        | SQLERRD.6
SAY 'SQLWARN ='SQLWARN.0',',
        | SQLWARN.1',',
        | SQLWARN.2',',
        | SQLWARN.3',',
        | SQLWARN.4',',
        | SQLWARN.5',',
        | SQLWARN.6',',
        | SQLWARN.7',',
        | SQLWARN.8',',
        | SQLWARN.9',',
        | SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
EXIT

```

Related concepts:

“REXX stored procedures” on page 650

Preparing a client program that calls a remote stored procedure

If you call a remote stored procedure from an embedded SQL application, you need to do a few extra steps when you prepare the client program. You do not need to do any extra steps when you prepare the stored procedure.

For an ODBC or CLI application, ensure that the DB2 packages and plan that are associated with the ODBC driver are bound to DB2. These packages and plan must be bound before you can run your application.

To prepare a client program that calls a remote stored procedure:

1. Precompile, compile, and link-edit the client program on the local DB2 subsystem.

2. Bind the resulting DBRM into a package at the local DB2 subsystem by using the BIND PACKAGE command with the option DBPROTOCOL(DRDA).

Recommendation: If you have packages that contain SQL CALL statements that you bound before DB2 Version 6, rebind them in DB2 Version 6 or later to get better performance from those packages. Rebinding lets DB2 obtain some information from the catalog at bind time that it obtained at run time before Version 6. Therefore, after you rebind your packages, they run more efficiently because DB2 can do fewer catalog searches at run time.

3. Bind the same DBRM, the one for the client program, into a package at the remote location by using the BIND PACKAGE command and specifying a location name. If your client program needs to access multiple servers, bind the program at each server.

Example: Suppose that you want a client program to call a stored procedure at location LOCA. You precompile the program to produce DBRM A. Then you can use the following command to bind DBRM A into package collection COLLA at location LOCA:

```
BIND PACKAGE (LOCA.COLLA) MEMBER(A)
```

4. Bind all packages into a plan on the local DB2 subsystem. Specify the bind option DBPROTOCOL(DRDA).
5. Bind any stored procedures that run under DB2 ODBC on a remote DB2 database server as a package at the remote site. Those procedures do not need to be bound into the DB2 ODBC plan.

Related tasks:

 Binding DBRMs to create packages (DB2 Programming for ODBC)

Related reference:

 BIND PACKAGE (DSN) (DB2 Commands)

How DB2 determines which stored procedure to run

A procedure is uniquely identified by its name and its qualifying schema name. You can tell DB2 exactly which stored procedure to run by qualifying it with its schema name when you call it. Otherwise, DB2 determines which stored procedure to run.

However, if you do not qualify the stored procedure name, DB2 uses the following method to determine which stored procedure to run:

1. DB2 searches the list of schema names from the PATH bind option or the CURRENT PATH special register from left to right until it finds a schema name for which a stored procedure definition exists with the name in the CALL statement.

DB2 uses schema names from the PATH bind option for CALL statements of the following form:

```
CALL procedure-name
```

DB2 uses schema names from the CURRENT PATH special register for CALL statements of the following form:

```
CALL host-variable
```

2. When DB2 finds a stored procedure definition, DB2 executes that stored procedure if the following conditions are true:
 - The caller is authorized to execute the stored procedure.

- The stored procedure has the same number of parameters as in the CALL statement.

If both conditions are not true, DB2 continues to go through the list of schemas until it finds a stored procedure that meets both conditions or reaches the end of the list.

3. If DB2 cannot find a suitable stored procedure, it returns an SQL error code for the CALL statement.

Calling different versions of a stored procedure from a single application

You can call different versions of a stored procedure from the same application program, even though those versions all have the same load module name.

To call different versions of a stored procedure from a single application:

1. When you define each version of the stored procedure, use the same stored procedure name but different schema names, different COLLID values, and different WLM environments.
2. In the program that invokes the stored procedure, specify the unqualified stored procedure name in the CALL statement.
3. Use the SQL path to indicate which version of the stored procedure that the client program should call. You can choose the SQL path in several ways:
 - If the client program is not an ODBC or JDBC application, use one of the following methods:
 - Use the CALL *procedure-name* form of the CALL statement. When you bind plans or packages for the program that calls the stored procedure, bind one plan or package for each version of the stored procedure that you want to call. In the PATH bind option for each plan or package, specify the schema name of the stored procedure that you want to call.
 - Use the CALL *host-variable* form of the CALL statement. In the client program, use the SET PATH statement to specify the schema name of the stored procedure that you want to call.
 - If the client program is an ODBC or JDBC application, choose one of the following methods:
 - Use the SET PATH statement to specify the schema name of the stored procedure that you want to call.
 - When you bind the stored procedure packages, specify a different collection for each stored procedure package. Use the COLLID value that you specified when defining the stored procedure to DB2.
4. When you run the client program, specify the plan or package with the PATH value that matches the schema name of the stored procedure that you want to call.

For example, suppose that you want to write one program, PROGY, that calls one of two versions of a stored procedure named PROCX. The load module for both stored procedures is named SUMMOD. Each version of SUMMOD is in a different load library. The stored procedures run in different WLM environments, and the startup JCL for each WLM environment includes a STEPLIB concatenation that specifies the correct load library for the stored procedure module.

First, define the two stored procedures in different schemas and different WLM environments:

```

CREATE PROCEDURE TEST.PROCX(IN V1 INTEGER, OUT V2 CHAR(9))
  LANGUAGE C
  EXTERNAL NAME SUMMOD
  WLM ENVIRONMENT TESTENV;
CREATE PROCEDURE PROD.PROCX(IN V1 INTEGER, OUT V2 CHAR(9))
  LANGUAGE C
  EXTERNAL NAME SUMMOD
  WLM ENVIRONMENT PRODENV;

```

When you write CALL statements for PROCX in program PROGY, use the unqualified form of the stored procedure name:

```
CALL PROCX(V1,V2);
```

Bind two plans for PROGY. In one BIND statement, specify PATH(TEST). In the other BIND statement, specify PATH(PROD).

To call TEST.PROCX, execute PROGY with the plan that you bound with PATH(TEST). To call PROD.PROCX, execute PROGY with the plan that you bound with PATH(PROD).

Invoking multiple instances of a stored procedure

Your application program can issue multiple CALL statements to the same local or remote stored procedure. Assume that your stored procedure returns result sets and the calling application leaves those result sets open before the next call to that same stored procedure. In that case, each CALL statement invokes a unique instance of the stored procedure.

To invoke multiple instances of the following stored procedures, ensure that both the client and server are in DB2 Version 8 new-function mode or later:

- Remote stored procedures
- Local stored procedures that have SQL to access a remote site

When you invoke multiple instances of a stored procedure, each instance runs serially within the same DB2 thread and opens its own result sets. These multiple calls invoke multiple instances of any packages that are invoked while running the stored procedure. These instances are invoked at either the same or different level of nesting under one DB2 connection or thread.

For local stored procedures that issue remote SQL, instances of the applications are created at the remote server site. These instances are created regardless of whether result sets exist or are left open between calls.

If you call too many instances of a stored procedure or if you open too many cursors, DB2 storage shortages and EDM POOL FULL conditions might occur. If the stored procedure issues remote SQL statements to another DB2 server, these conditions can occur at both the DB2 client and at the DB2 server.

To invoke multiple instances of a stored procedure:

1. To optimize storage usage and prevent storage shortages, ensure that you specify appropriate values for the following two subsystem parameters:

MAX_ST_PROC

Controls the maximum number of stored procedure instances that you can call within the same thread.

MAX_NUM_CUR

Controls the maximum number of cursors that can be opened by the same thread.

When either of the values from these subsystem parameters is exceeded while an application is running, the CALL statement or the OPEN statement receives SQLCODE -904.

2. In your application, issue CALL statements to the stored procedure.
3. In the calling application for the stored procedure, close the result sets and issue frequent commits. Even read-only applications should perform these actions.

Applications that fail to close result sets or issue an adequate number of commits might terminate abnormally with DB2 storage shortage and EDM POOL FULL conditions.

Related reference:

[➤](#) MAX OPEN CURSORS field (MAX_NUM_CUR subsystem parameter) (DB2 Installation and Migration)

[➤](#) MAX STORED PROCS field (MAX_ST_PROC subsystem parameter) (DB2 Installation and Migration)

[➤](#) CALL (DB2 SQL)

Designating the active version of a native SQL procedure

When a native SQL procedure is called, DB2 uses the version that is designated as the active version.

When you create a native SQL procedure, that first version is by default the active version. If you create additional versions of a stored procedure, you can designate another version to be the active version.

Exception: If an existing active version is still being used by a process, the new active version is not used until the next call to that procedure.

To designate the active version of a native SQL procedure, issue an ALTER PROCEDURE statement with the following items:

- The name of the native SQL procedure for which you want to change the active version.
- The ACTIVATE VERSION clause with the name of the version that you want to be active.

When the ALTER statement is committed, the new version of the procedure becomes the active version and is used by the next call for that procedure.

Example: The following ALTER PROCEDURE statement makes version V2 of the UPDATE_BALANCE procedure the active version.

```
ALTER PROCEDURE UPDATE_BALANCE
  ACTIVATE VERSION V2;
```

Temporarily overriding the active version of a native SQL procedure

If you want a particular call to a native SQL procedure to use a version other than the active version, you can temporarily override the active version. Such an override might be helpful when you are testing a new version of a native SQL procedure.

Recommendation: If you want all calls to a native SQL procedure to use a particular version, do not temporarily override the active version in every call. Instead, make that version the active version. Otherwise, performance might be slower.

To temporarily override the active version of a native SQL procedure, specify the following statements in your program:

1. The SET CURRENT ROUTINE VERSION statement with the name of the version of the procedure that you want to use. If the specified version does not exist, the active version is used.
2. The CALL statement with the name of the procedure.

The following CALL statement invokes version V1 of the UPDATE_BALANCE procedure, regardless of what the current active version of that procedure is.

```
SET CURRENT ROUTINE VERSION = V1;  
SET procname = 'UPDATE_BALANCE';  
CALL :procname USING DESCRIPTOR :x;
```

Specifying the number of stored procedures that can run concurrently

Multiple stored procedures can run concurrently, each under its own z/OS task control block (TCB). The z/OS Workload Manager (WLM) manages how many concurrent stored procedures can run in an address space. The number of concurrent stored procedures in an address space cannot exceed the value of the NUMTCB field that was specified on the DSNTIPX installation panel, during DB2 installation.

You can override that value in the following ways:

- Edit the JCL procedures that start stored procedures address spaces, and modify the value of the NUMTCB parameter.
- Specify the following parameter in the Start Parameters field of the Create An Application Environment panel when you set up a WLM application environment:

```
NUMTCB=number-of-TCBs
```

Special cases:

- For REXX stored procedures, you must set the NUMTCB parameter to 1.
- Stored procedures that invoke utilities can invoke only one utility at a time in a single address space. Consequently, the value of the NUMTCB parameter is forced to 1 for those procedures.

Related concepts:

 Installation step 19: Configure DB2 for running stored procedures and user-defined functions (DB2 Installation and Migration)

 Migration step 25: Configure DB2 for running stored procedures and user-defined functions (DB2 Installation and Migration)

Related tasks:

 Maximizing the number of procedures or functions that run in an address space (DB2 Performance)

Retrieving the procedure status

When an SQL procedure returns control to the calling program, it also returns the procedure status. The status is an integer value that indicates the success of the procedure.

DB2 sets the status to 0 or -1 depending on the value of the SQLCODE. Alternatively, an SQL procedure can set the integer status value by using the RETURN statement. In this case, DB2 sets the SQLCODE in the SQLCA to 0.

To retrieve the procedure status, perform one of the following actions in the calling program:

- Issue the GET DIAGNOSTICS statement with the DB2_RETURN_STATUS item. The specified host variable in the GET DIAGNOSTICS statement is set to one of the following values:
 - 0 This value indicates that the procedure returned with an SQLCODE that is greater or equal to zero. You can access the value directly from the SQLCA by retrieving the value of SQLERRD(1). For C applications, retrieve SQLERRD[0].
 - 1 This value indicates that the procedure returned with an SQLCODE that is less than zero. In this case, the SQLERRD(1) value in the SQLCA is not set. DB2 returns -1 only.
 - n* Any value other than 0 or -1 is the return value that was explicitly set in the procedure with the RETURN statement.

Example of using GET DIAGNOSTICS to retrieve the return status: The following SQL code creates an SQL procedure that is named TESTIT, which calls another SQL procedure that is named TRYIT. The TRYIT procedure returns a status value. The TESTIT procedure retrieves that value with the DB2_RETURN_STATUS item of the GET DIAGNOSTICS statement.

```
CREATE PROCEDURE TESTIT ()
LANGUAGE SQL
A1:BEGIN
  DECLARE RETVAL INTEGER DEFAULT 0;
  ...
  CALL TRYIT;
  GET DIAGNOSTICS RETVAL = DB2_RETURN_STATUS;
  IF RETVAL <> 0 THEN
    ...
    LEAVE A1;
  ELSE
    ...
  END IF;
END A1
```

- Retrieve the value of SQLERRD(1) in the SQLCA. For C applications, retrieve SQLERRD[0]. This field contains the integer value that was set by the RETURN statement in the SQL procedure. This method is not applicable if the status was set by DB2.

Related concepts:

 SQL communication area (SQLCA) (DB2 SQL)

Related reference:

 GET DIAGNOSTICS (DB2 SQL)

Writing a program to receive the result sets from a stored procedure

You can write a program to receive results set from a stored procedure for either a fixed number of result sets, for which you know the contents, or a variable number of result sets, for which you do not know the contents.

A program for a fixed number of result sets is simpler to write than a program for a variable number of result sets. However, if you write a program for a variable number of result sets, you do not need to make modifications to the program if the stored procedure changes.

If your program calls an SQL procedure that returns result sets, you must write the program for a fixed number of result sets.

In the following steps, you do not need to connect to the remote location when you execute these statements:

- DESCRIBE PROCEDURE
- ASSOCIATE LOCATORS
- ALLOCATE CURSOR
- DESCRIBE CURSOR
- FETCH
- CLOSE

To write a program to receive the result sets from a stored procedure:

1. Declare a locator variable for each result set that is to be returned.
If you do not know how many result sets are to be returned, declare enough result set locators for the maximum number of result sets that might be returned.
2. Call the stored procedure and check the SQL return code.
If the SQLCODE from the CALL statement is +466, the stored procedure has returned result sets.
3. Determine how many result sets the stored procedure is returning.
If you already know how many result sets the stored procedure returns, skip this step.

Use the SQL statement DESCRIBE PROCEDURE to determine the number of result sets. DESCRIBE PROCEDURE places information about the result sets in an SQLDA. Make this SQLDA large enough to hold the maximum number of result sets that the stored procedure might return. When the DESCRIBE PROCEDURE statement completes, the fields in the SQLDA contain the following values:

- SQLD contains the number of result sets that are returned by the stored procedure.
 - Each SQLVAR entry gives the following information about a result set:
 - The SQLNAME field contains the name of the SQL cursor that is used by the stored procedure to return the result set.
 - The SQLIND field contains the value -1, which indicates that no estimate of the number of rows in the result set is available.
 - The SQLDATA field contains the value of the result set locator, which is the address of the result set.
4. Link result set locators to result sets by performing one of the following actions:

- Use the ASSOCIATE LOCATORS statement. You must embed this statement in an application or SQL procedure. The ASSOCIATE LOCATORS statement assigns values to the result set locator variables. If you specify more locators than the number of result sets that are returned, DB2 ignores the extra locators.
- If you executed the DESCRIBE PROCEDURE statement previously, the result set locator values are in the SQLDATA fields of the SQLDA. You can copy the values from the SQLDATA fields to the result set locators manually, or you can execute the ASSOCIATE LOCATORS statement to do it for you.

The stored procedure name that you specify in an ASSOCIATE LOCATORS statement or DESCRIBE PROCEDURE statement must match the stored procedure name in the CALL statement as follows:

- If the name is unqualified in the CALL statement, do not qualify it.
- If the name is qualified with a schema name in the CALL statement, qualify it with the schema name.
- If the name is qualified with a location name and schema name in the CALL statement, qualify it with a location name and schema name.

5. Allocate cursors for fetching rows from the result sets.

Use the SQL statement ALLOCATE CURSOR to link each result set with a cursor. Execute one ALLOCATE CURSOR statement for each result set. The cursor names can differ from the cursor names in the stored procedure.

To use the ALLOCATE CURSOR statement, you must embed it in an application or SQL procedure.

6. Determine the contents of the result sets.

If you already know the format of the result set, skip this step.

Use the SQL statement DESCRIBE CURSOR to determine the format of a result set and put this information in an SQLDA. For each result set, you need an SQLDA that is big enough to hold descriptions of all columns in the result set.

You can use DESCRIBE CURSOR for only those cursors for which you executed ALLOCATE CURSOR previously.

After you execute DESCRIBE CURSOR, if the cursor for the result set is declared WITH HOLD, the high-order bit of byte 8 of field SQLDAID in the SQLDA is set to 1.

7. Fetch rows from the result sets into host variables by using the cursors that you allocated with the ALLOCATE CURSOR statements. Fetching rows from a result set is the same as fetching rows from a table.

If you executed the DESCRIBE CURSOR statement, perform the following steps before you fetch the rows:

- Allocate storage for host variables and indicator variables. Use the contents of the SQLDA from the DESCRIBE CURSOR statement to determine how much storage you need for each host variable.
- Put the address of the storage for each host variable in the appropriate SQLDATA field of the SQLDA.
- Put the address of the storage for each indicator variable in the appropriate SQLIND field of the SQLDA.

The following examples show C language code that accomplishes each of these steps. Coding for other languages is similar.

The following example demonstrates how to receive result sets when you know how many result sets are returned and what is in each result set.

```

/*****/
/* Declare result set locators. For this example, */
/* assume you know that two result sets will be returned. */
/* Also, assume that you know the format of each result set. */
/*****/
EXEC SQL BEGIN DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2;
EXEC SQL END DECLARE SECTION;

:
/*****/
/* Call stored procedure P1. */
/* Check for SQLCODE +466, which indicates that result sets */
/* were returned. */
/*****/
EXEC SQL CALL P1(:parm1, :parm2, ..);
if(SQLCODE==+466)
{
/*****/
/* Establish a link between each result set and its */
/* locator using the ASSOCIATE LOCATORS. */
/*****/
EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2) WITH PROCEDURE P1;

:
/*****/
/* Associate a cursor with each result set. */
/*****/
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
/*****/
/* Fetch the result set rows into host variables. */
/*****/
while(SQLCODE==0)
{
EXEC SQL FETCH C1 INTO :order_no, :cust_no;

:
}
while(SQLCODE==0)
{
EXEC SQL FETCH C2 :order_no, :item_no, :quantity;

:
}
}

```

The following example demonstrates how to receive result sets when you do not know how many result sets are returned or what is in each result set.

```

/*****/
/* Declare result set locators. For this example, */
/* assume that no more than three result sets will be */
/* returned, so declare three locators. Also, assume */
/* that you do not know the format of the result sets. */
/*****/
EXEC SQL BEGIN DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
EXEC SQL END DECLARE SECTION;

:
/*****/
/* Call stored procedure P2. */
/* Check for SQLCODE +466, which indicates that result sets */
/* were returned. */
/*****/

```

```

EXEC SQL CALL P2(:parm1, :parm2, ...);
if(SQLCODE==+466)
{
/*****/
/* Determine how many result sets P2 returned, using the */
/* statement DESCRIBE PROCEDURE. :proc_da is an SQLDA */
/* with enough storage to accommodate up to three SQLVAR */
/* entries. */
/*****/
EXEC SQL DESCRIBE PROCEDURE P2 INTO :proc_da;

:
/*****/
/* Now that you know how many result sets were returned, */
/* establish a link between each result set and its */
/* locator using the ASSOCIATE LOCATORS. For this example, */
/* we assume that three result sets are returned. */
/*****/
EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P2;

:
/*****/
/* Associate a cursor with each result set. */
/*****/
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
EXEC SQL ALLOCATE C3 CURSOR FOR RESULT SET :loc3;

/*****/
/* Use the statement DESCRIBE CURSOR to determine the */
/* format of each result set. */
/*****/
EXEC SQL DESCRIBE CURSOR C1 INTO :res_da1;
EXEC SQL DESCRIBE CURSOR C2 INTO :res_da2;
EXEC SQL DESCRIBE CURSOR C3 INTO :res_da3;

:
/*****/
/* Assign values to the SQLDATA and SQLIND fields of the */
/* SQLDAs that you used in the DESCRIBE CURSOR statements. */
/* These values are the addresses of the host variables and */
/* indicator variables into which DB2 will put result set */
/* rows. */
/*****/

:
/*****/
/* Fetch the result set rows into the storage areas */
/* that the SQLDAs point to. */
/*****/
while(SQLCODE==0)
{
EXEC SQL FETCH C1 USING :res_da1;

:
}
while(SQLCODE==0)
{
EXEC SQL FETCH C2 USING :res_da2;

:
}
while(SQLCODE==0)
{
EXEC SQL FETCH C3 USING :res_da3;

```

```

:
}
}

```

The following example demonstrates how you can use an SQL procedure to receive result sets. The logic assumes that no handler exists to intercept the +466 SQLCODE, such as `DECLARE CONTINUE HANDLER FOR SQLWARNING`. Such a handler causes SQLCODE to be reset to zero. Then the test for `IF SQLCODE = 466` is never true and the statements in the IF body are never executed.

```







DECLARE RESULT1 RESULT_SET_LOCATOR VARYING;
DECLARE RESULT2 RESULT_SET_LOCATOR VARYING;
DECLARE AT_END, VAR1, VAR2 INT DEFAULT 0;
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET AT_END = 99;
SET TOTAL1 = 0;
SET TOTAL2 = 0;
CALL TARGETPROCEDURE();
IF SQLCODE = 466 THEN
  ASSOCIATE RESULT SET LOCATORS(RESULT1,RESULT2)
  WITH PROCEDURE SPDG3091;
  ALLOCATE RSCUR1 CURSOR FOR RESULT1;
  ALLOCATE RSCUR2 CURSOR FOR RESULT2;
  WHILE AT_END = 0 DO
    FETCH RSCUR1 INTO VAR1;
    SET TOTAL1 = TOTAL1 + VAR1;
    SET VAR1 = 0; /* Reset so the last value fetched is not added after AT_END */
  END WHILE;
  SET AT_END = 0; /* Reset for next loop */
  WHILE AT_END = 0 DO
    FETCH RSCUR2 INTO VAR2;
    SET TOTAL2 = TOTAL2 + VAR2;
    SET VAR2 = 0; /* Reset so the last value fetched is not added after AT_END */
  END WHILE;
END IF;

```

Related concepts:

“Examples of programs that call stored procedures” on page 252

Related reference:

-  [ALLOCATE CURSOR \(DB2 SQL\)](#)
-  [ASSOCIATE LOCATORS \(DB2 SQL\)](#)
-  [CALL \(DB2 SQL\)](#)
-  [DESCRIBE CURSOR \(DB2 SQL\)](#)
-  [DESCRIBE PROCEDURE \(DB2 SQL\)](#)
-  [SQL descriptor area \(SQLDA\) \(DB2 SQL\)](#)

DB2-supplied stored procedures

DB2 provides some stored procedures that you can call in your application programs to perform a number of utility and application programming functions. Typically, these procedures are created during installation or migration.

The following table lists all of the DB2-supplied stored procedures.

Table 132. DB2-supplied stored procedures

Stored procedure name	Description
ADMIN_COMMAND_DB2	The ADMIN_COMMAND_DB2 stored procedure is an administrative enablement routine. It executes one or more DB2 commands on a connected DB2 subsystem or on a DB2 data sharing group member and returns the command output messages.
ADMIN_COMMAND_DSN	The ADMIN_COMMAND_DSN stored procedure is an administrative enablement routine. It executes a BIND, REBIND, or FREE DSN subcommand and then returns the output from the subcommand.
ADMIN_COMMAND_UNIX	The ADMIN_COMMAND_UNIX stored procedure is an administrative enablement routine. It executes a USS command and returns the output from the command.
ADMIN_DS_BROWSE	<p>The ADMIN_DS_BROWSE stored procedure is an administrative enablement routine. It returns either text or binary records from one of the following entities:</p> <ul style="list-style-type: none"> • a physical sequential (PS) data set • a generation data set (GDS) • a partitioned data set (PDS) • a partitioned data set extended (PDSE) member. <p>This stored procedure supports only data sets with LRECL=80 and RECFM=FB.</p>
ADMIN_DS_DELETE	<p>The ADMIN_DS_DELETE stored procedure is an administrative enablement routine. It deletes one of the following entities:</p> <ul style="list-style-type: none"> • a physical sequential (PS) data set • a partitioned data set (PDS) • a partitioned data set extended (PDSE) • a generation data set (GDS) • a member of a PDS or PDSE
ADMIN_DS_LIST	<p>The ADMIN_DS_LIST stored procedure is an administrative enablement routine. It returns a list of one of the following items:</p> <ul style="list-style-type: none"> • data set names • generation data groups (GDG) • partitioned data set (PDS) members • partitioned data set extended (PDSE) members • generation data sets of a GDG
ADMIN_DS_RENAME	<p>The ADMIN_DS_RENAME stored procedure is an administrative enablement routine. It renames one of the following entities:</p> <ul style="list-style-type: none"> • a physical sequential (PS) data set • a partitioned data set (PDS) • a partitioned data set extended (PDSE) • a member of a PDS or PDSE

Table 132. DB2-supplied stored procedures (continued)

Stored procedure name	Description
ADMIN_DS_SEARCH	<p>The ADMIN_DS_SEARCH stored procedure is an administrative enablement routine. It determines if one of the following items is cataloged:</p> <ul style="list-style-type: none"> • a physical sequential (PS) data set • a partitioned data set (PDS) • a partitioned data set extended (PDSE) • a generation data group (GDG) • a generation data set (GDS) <p>Alternatively, ADMIN_DS_SEARCH determines if a library member of a cataloged PDS or PDSE exists.</p>
ADMIN_DS_WRIT	<p>The ADMIN_DS_WRITE stored procedure is an administrative enablement routine. It writes either text or binary records that are passed in a global temporary table to one of the following entities:</p> <ul style="list-style-type: none"> • a physical sequential (PS) data set • partitioned data set (PDS) member • partitioned data set extended (PDSE) member • generation data set (GDS) <p>ADMIN_DS_WRITE can either append or replace an existing PS data set, PDS member, PDSE member, or GDS. ADMIN_DS_WRITE can create one of the following entities:</p> <ul style="list-style-type: none"> • a PS data set • PDS data set or member • PDSE data set or member • GDS for an existing generation data group (GDG) as needed <p>This stored procedure supports only data sets with LRECL=80 and RECFM=FB.</p>
ADMIN_INFO_HOST	<p>The ADMIN_INFO_HOST stored procedure is an administrative enablement routine. It returns the host name of a connected DB2 subsystem or the host name of every member of a data sharing group.</p>
ADMIN_INFO_SSID	<p>The ADMIN_INFO_SSID stored procedure is an administrative enablement routine. It returns the subsystem ID of the connected DB2 subsystem.</p>
ADMIN_JOB_CANCEL	<p>The ADMIN_JOB_CANCEL stored procedure is an administrative enablement routine. It purges or cancels a job.</p>
ADMIN_JOB_FETCH	<p>The ADMIN_JOB_FETCH stored procedure is an administrative enablement routine. It retrieves the output from the JES spool.</p>
ADMIN_JOB_QUERY	<p>The ADMIN_JOB_QUERY stored procedure is an administrative enablement routine. It displays the status and completion information of a job.</p>
ADMIN_JOB_SUBMIT	<p>The ADMIN_JOB_SUBMIT stored procedure is an administrative enablement routine. It submits a job to a JES2 or JES3 system.</p>
ADMIN_TASK_ADD	<p>The ADMIN_TASK_ADD stored procedure is an administrative task scheduler routine. It adds a task to the task list of the administrative task scheduler.</p>
ADMIN_TASK_REMOVE	<p>The ADMIN_TASK_REMOVE stored procedure is an administrative task scheduler routine. It removes a task from the task list of the administrative task scheduler.</p>

Table 132. DB2-supplied stored procedures (continued)

Stored procedure name	Description
ADMIN_UTL_SCHEDULE	The ADMIN_UTL_SCHEDULE stored procedure is an administrative enablement routine. It executes utilities in parallel.
ADMIN_UTL_SORT	The ADMIN_UTL_SORT stored procedure is an administrative enablement routine. It sorts database objects for parallel utility execution using JCL or the ADMIN_UTL_SCHEDULE stored procedure.
DSNACCOR	The real-time statistics stored procedure, DSNACCOR, queries the DB2 real-time statistics tables. This information helps you determine when you should run COPY, REORG, or RUNSTATS utility jobs, or enlarge your DB2 data sets.
DSNACCOX	The enhanced DB2 real-time statistics stored procedure, DSNACCOX, makes recommendations to help you maintain your DB2 databases. The DSNACCOX stored procedure replaces the previous DSNACCOR stored procedure, beginning in Version 9.
DSNACICS	The CICS transaction invocation stored procedure, DSNACICS, invokes CICS transactions from a remote workstation.
DSNAEXP	The DB2 EXPLAIN stored procedure, DSN8EXP, invokes the EXPLAIN function on an SQL statement without requiring you to have the authorization to execute that SQL statement. The DSNAEXP stored procedure replaces the previous DSN8EXP stored procedure, beginning in Version 8. DSN8EXP handles SQL statements of up to 32,700 bytes in length. DSNAEXP can handle longer statements.
DSNAHVPM	The DSNAHVPM stored procedure is used by Optimization Service Center for DB2 for z/OS to convert host variables in a static SQL statement to typed parameter markers.
DSNAIMS	The IMS transactions stored procedure, DSNAIMS, invokes IMS transactions and commands, without requiring a DB2 subsystem to maintain its own connection to IMS.
DSNAIMS2	The IMS transactions stored procedure 2, DSNAIMS2, performs the same function as DSNAIMS, except that DSNAIMS2 also includes multi-segment input support for IMS transactions.
DSNLEUSR	The SYSIBM.USERNAMES encryption stored procedure, DSNLEUSR, stores encrypted values in the NEWAUTHID and PASSWORD fields of the SYSIBM.USERNAMES catalog table.
DSNTBIND	The DSNTBIND stored procedure binds Java stored procedures.
DSNTPSMP	The DB2 for z/OS SQL procedure processor, DSNTPSMP, is a REXX stored procedure that prepares external SQL procedures for execution.
DSNUTILS	The utilities stored procedure for EBCDIC input, DSNUTILS, invokes DB2 utilities from a local or remote client program. This stored procedure accepts utility control statements that are encoded in EBCDIC.
DSNUTILU	The utilities stored procedure for Unicode input, DSNUTILU, invokes DB2 utilities from a local or remote client program. This stored procedure accepts utility control statements that are encoded in Unicode.
DSNWSPM	The DSNWSPM stored procedure formats IFCID 148 records.

Table 132. DB2-supplied stored procedures (continued)

Stored procedure name	Description
DSNWZP	The subsystem parameter stored procedure, DSNWZP, is used by the DB2-supplied stored procedure WLM_REFRESH.
GET_CONFIG	<p>The GET_CONFIG stored procedure is a common SQL API stored procedure. It returns information about the data server configuration, including information about the following items:</p> <ul style="list-style-type: none"> • the data sharing group • the DB2 subsystem parameters • the DDF status and configuration • the connected DB2 subsystem • the RLF tables • the active log data sets • the last DB2 restart <p>This stored procedure is used primarily by DB2 tools.</p>
GET_MESSAGE	The GET_MESSAGE stored procedure is a common SQL API stored procedure. It returns the short message text for an SQL code. This stored procedure is used primarily by DB2 tools.
GET_SYSTEM_INFO	<p>The GET_SYSTEM_INFO stored procedure is a common SQL API stored procedure. It returns system information, including information about the following items:</p> <ul style="list-style-type: none"> • operating system • product information • PTF level of each DB2 module • the SMP/E APPLY status of the requested SYSMOD • WLM classification rules that apply to the DB2 workload for subsystem types DB2 and DDF <p>This stored procedure is used primarily by DB2 tools</p>
SQLJ.ALTER_JAVA_PATH	The SQLJ.ALTER_JAVA_PATH stored procedure specifies the class resolution path that the JVM searches to resolve class references. This action is needed if a JAR that you have installed refers to classes in other installed JARs.
SQLJ.DB2_INSTALL_JAR	The SQLJ.DB2_INSTALL_JAR stored procedure installs a set of Java classes into a local or remote catalog.
SQLJ.DB2_REMOVE_JAR	The SQLJ.DB2_REMOVE_JAR stored procedure removes a Java JAR file and its classes from a local or remote catalog.
SQLJ.DB2_REPLACE_JAR	The SQLJ.DB2_REPLACE_JAR stored procedure replaces a previously installed JAR file in a local or remote catalog.
SQLJ.DB2_UPDATEJARINFO	The SQLJ.DB2_UPDATEJARINFO stored procedure inserts class, class source, and associated options for a previously installed JAR file in a local or remote catalog.
SQLJ.INSTALL_JAR	The SQLJ.INSTALL_JAR stored procedure installs a set of Java classes into the current SQL catalog and schema.
SQLJ.REMOVE_JAR	The SQLJ.REMOVE_JAR stored procedure removes a Java JAR file and its classes from a specified, local catalog.
SQLJ.REPLACE_JAR	The SQLJ.REPLACE_JAR stored procedure replaces a previously installed JAR file in a local catalog.
WLM_REFRESH	The WLM environment refresh stored procedure, WLM_REFRESH, refreshes a WLM environment from a remote workstation.

Table 132. DB2-supplied stored procedures (continued)

Stored procedure name	Description
WLM_SET_CLIENT_INFO	The WLM_SET_CLIENT_INFO stored procedure sets client information that is associated with the current connection at the DB2 server.
XSR_ADDSCHEMADOC	The add XML schema document stored procedure, XSR_ADDSCHEMADOC, adds every XML schema other than the primary XML schema document to the XSR.
XSR_COMPLETE	The XML schema registration completion stored procedure, XSR_COMPLETE, is the final stored procedure to be called as part of the XML schema registration process. The XML schema registration process registers XML schemas with the XSR.
XSR_REGISTER	The XML schema registration stored procedure, XSR_REGISTER, is the first stored procedure to be called as part of the XML schema registration process. The XML schema registration process registers XML schemas with the XSR.
XSR_REMOVE	The XML schema removal stored procedure, XSR_REMOVE, removes all components of an XML schema.

Related reference:

- [➤ DB2-supplied stored procedures and user-defined functions \(DB2 Installation and Migration\)](#)
- [➤ Source code for activating DB2-supplied stored procedures \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

Related information:

- [➤ Stored procedures for administration \(DB2 Administration Guide\)](#)

WLM_REFRESH stored procedure

The WLM_REFRESH stored procedure refreshes a WLM environment. This stored procedure can recycle the environment in which it runs and in any other WLM environment.

Environment for WLM_REFRESH

The load module for WLM_REFRESH, DSNTWR, must reside in an APF-authorized library.

| DSNTWR runs module DSNTWRE as a subtask. DSNTWRE retrieves a copy of the
 | DB2 environment information block (EIB) for the DB2 subsystem. The DSNTWRE
 | module has no APF-authorization requirement, but it must reside in an
 | APF-authorized library to be available to DSNTWR.

| **Recommendation:** Use WLM environment DSNWLM_GENERAL for running
 | WLM_REFRESH. Installation job DSNTIJMV creates an address space procedure
 | called DSNWLMG for this environment.

Authorization required for WLM_REFRESH

To execute the CALL statement, the SQL authorization ID of the process must have READ access or higher to the z/OS Security Server System Authorization Facility (SAF) resource profile *ssid.WLM_REFRESH.WLM-environment-name* in resource class DSNR. This is a different resource profile from the *ssid.WLMENV.WLM-*

environment-name resource profile, which DB2 uses to determine whether a stored procedure or user-defined function is authorized to run in the specified WLM environment.

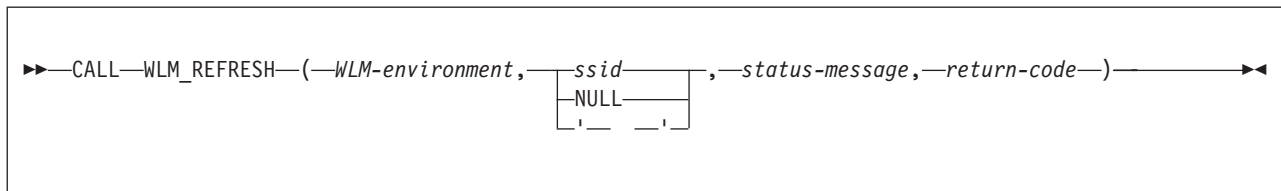
WLM_REFRESH uses an extended MCS console to monitor the operating system response to a WLM environment refresh request. The privilege to create an extended MCS console is controlled by the resource profile MVS.MCSOPER.* in the OPERCMDS class. If the MVS.MCSOPER.* profile exists, or if the specific profile MVS.MCSOPER.DSNTWR exists, the task ID that is associated with the WLM environment in which WLM_REFRESH runs must have READ access to it.

If the MVS.VARY.* profile exists, or if the specific profile MVS.VARY.WLM exists, the task ID that is associated with the WLM environment in which WLM_REFRESH runs must have CONTROL access to it.

WLM_REFRESH syntax diagram

The WLM_REFRESH stored procedure refreshes a WLM environment. WLM_REFRESH can recycle the environment in which it runs, as well as any other WLM environment.

The following syntax diagram shows the SQL CALL statement for invoking WLM_REFRESH. The linkage convention for WLM_REFRESH is GENERAL WITH NULLS.



WLM_REFRESH option descriptions

WLM-environment

Specifies the name of the WLM environment that you want to refresh. This is an input parameter of type VARCHAR(32).

ssid

Specifies the subsystem ID of the DB2 subsystem with which the WLM environment is associated. If this parameter is NULL or blank, DB2 uses one of the following values for this parameter:

- In a non-data sharing environment, DB2 uses the subsystem ID of the subsystem on which WLM_REFRESH runs.
- In a data sharing environment, DB2 uses the group attach name for the data sharing group in which WLM_REFRESH runs.

This is an input parameter of type VARCHAR(4).

status-message

Contains an informational message about the execution of the WLM refresh. This is an output parameter of type VARCHAR(120).

return-code

Contains the return code from the WLM_REFRESH call, which is one of the following values:

- 0 WLM_REFRESH executed successfully.

- 4 One of the following conditions exists:
 - The SAF resource profile *ssid.WLM_REFRESH.wlm-environment* is not defined in resource class DSNR.
 - The SQL authorization ID of the process (CURRENT SQLID) is not defined to SAF.
 - The wait time to obtain a response from z/OS was exceeded.
- 8 The SQL authorization ID of the process (CURRENT SQLID) is not authorized to refresh the WLM environment.
- 990 DSNTWR received an unexpected SQLCODE while determining the current SQLID.
- 993 One of the following conditions exists:
 - The *WLM-environment* parameter value is null, blank, or contains invalid characters.
 - The *ssid* value contains invalid characters.
- 994 The extended MCS console was not activated within the number of seconds indicated by message DSNT5461.
- 995 DSNTWR is not running as an authorized program.
- 996 DSNTWR could not activate an extended MCS console. See message DSNT533I for more information.
- 997 DSNTWR made an unsuccessful request for a message from its extended MCS console. See message DSNT533I for more information.
- 998 The extended MCS console for DSNTWR posted an alert. See message DSNT534I for more information.
- 999 The operating system denied an authorized WLM_REFRESH request. See message DSNT545I for more information.

return-code is an output parameter of type INTEGER.

Example of WLM_REFRESH invocation

Suppose that you want to refresh WLM environment WLMENV1, which is associated with a DB2 subsystem with ID DSN. Assume that you already have READ access to the DSN.WLM_REFRESH.WLMENV1 SAF profile. The CALL statement for WLM_REFRESH looks like this:

```
strcpy(WLMENV,"WLMENV1");
strcpy(SSID,"DSN");
EXEC SQL CALL SYSPROC.WLM_REFRESH(:WLMENV, :SSID, :MSGTEXT, :RC);
```

For a complete example of setting up access to an SAF profile and calling WLM_REFRESH, see job DSNTJ6W, which is in data set *prefix.SDSNSAMP*.

Related information:

 Controlling Extended MCS Consoles Using RACF (z/OS MVS Planning: Operations)

DSNTWRE program

The DSNTWRE program retrieves a copy of the DB2 environment information block (EIB) for a DB2 subsystem.

DSNTWRE is primarily for use by stored procedure WLM_REFRESH, but can be called by any program that needs the contents of the EIB. DSNTWRE can be attached as a subtask or linked and run inline.

DSNTWRE is available as source code, in data set member *prefix.SDSNSAMP(DSNTWRE)*, or as an executable module, in data set member *prefix.SDSNLOAD(DSNTWRE)*.

DSNTWRE parameters

The parameter list for DSNTWRE is as follows:

Table 133. DSNTWRE parameter list

Parameter number	Data type and length	Input value	Output value
1	Character, 4 bytes	DB2 subsystem name.	Unchanged.
2	Pointer to a 4-byte integer	Not used.	Return code from DSNTWRE.
3	Pointer to a character field that is large enough to hold the EIB	Not used.	The EIB, if the return code from DSNTWRE is 0. Unchanged otherwise.
4	Pointer to a 120-byte, varying-length character field	Not used.	An error message, if the return code from DSNTWRE is not 0. Unchanged otherwise.

Example of invoking DSNTWRE

The following C application calls DSNTWRE, and prints the contents of the EIB for a DB2 subsystem with subsystem ID DSN.

```
#pragma      linkage( DSNTWRE,OS ) /* DSNTWRE: Gets DB2's EIB */
#include      <stdio.h>
int main( int argc, char *argv[] )
{
    char      ssID[5] = "DSN "; /* DB2 subsystem name */
    int       ret; /* DSNTWRE return code */
    struct    /* Area for DB2 EIB */
    { short int EIBCODE; /* EIB block identifier */
      short int EIBTLEN; /* EIB block length */
      char      EIBEYEC[4]; /* Eye catcher "EIB " */
      char      EIBSSID[4]; /* DB2 subsystem ID */
      char      EIBGATT[4]; /* DB2 group attach name
                           /* or blanks */
      char      EIBGRPN[8]; /* Data sharing group name
                           /* or blanks */
      char      EIBMBRN[8]; /* Data sharing member name
                           /* or blanks */
      char      EIBXSEC; /* DSNX@XAC/SECLABEL class
                           /* active Y/N */
      char      EIBFLAGS; /* Flag byte */
      char      EIBSGATT[4]; /* DB2 subgroup attach name */
      char      EIBRSV1[2]; /* Reserved */
      char      EIBRSV2[8]; /* Reserved */
    } EIB; /* DB2 Environment Info Block */
    struct    /* DSNTWRE message area */
    { short int msgLen; /* Message length */
```

```

        char    msgTxt[120];        /* Message text          */
    }          msg;                /* DSNTWRE output message */
char    X80   = '\x80';          /* Bit mask              */
char    X40   = '\x40';          /* Bit mask              */

/* Call DSNTWRE *****/
DSNTWRE( &ssID, &ret, &EIB, &msg );

if( ret == 0 )
{
    printf( "DSNTWRE call succeeded - "
            "DB2 Environment Information Block follows\n" );
    printf( "* Subsystem ID          = %.4s\n", EIB.EIBSSID );
    printf( "* Group name              = %.4s\n", EIB.EIBGRPN );
    printf( "* Member name           = %.4s\n", EIB.EIBMBRN );
    printf( "* Group attach name      = %.8s\n", EIB.EIBGATT );
    printf( "* Subgroup attach name  = %.8s\n", EIB.EIBSGATT );
    printf( "* SECLEVEL class active = %c\n", EIB.EIBXSEC );
    if( (EIB.EIBFLAGS & X80) == X80 )
        printf( "* RESTART LIGHT INDOUBTS = Y\n" );
    else
        printf( "* RESTART LIGHT INDOUBTS = N\n" );
    if( (EIB.EIBFLAGS & X40) == X40 )
        printf( "* New function mode      = Y\n" );
    else
        printf( "* New function mode      = N\n" );
}
else
{
    printf( "DSNTWRE call failed - Diagnostics follow\n" );
    printf( "* DSNTWRE return code = %i\n", ret );
    printf( "* DSNTWRE message   = %.120s\n", msg.msgTxt );
}
return( ret );

} /* End of main */

```

WLM_SET_CLIENT_INFO stored procedure

This procedure allows the caller to set client information that is associated with the current connection at the DB2 for z/OS server.

The following DB2 for z/OS client special registers can be changed:

- CURRENT CLIENT_ACCTNG
- CURRENT CLIENT_USERID
- CURRENT CLIENT_WRKSTNNAME
- CURRENT CLIENT_APPLNAME

The existing behavior of the CLIENT_ACCTNG register is unchanged. It gets its value from the accounting token for DSN requesters, and from the accounting string for SQL and other requesters.

This procedure is not under transaction control and client information changes made by the procedure are independent of committing or rolling back units of work.

Environment

WLM_SET_CLIENT_INFO runs in a WLM-established stored procedures address space.

Authorization

To execute the CALL statement, the owner of the package or plan that contains the CALL statement must have one or more of the following privileges on each package that the stored procedure uses:

- The EXECUTE privilege on the package for DSNADMSI
- Ownership of the package
- PACKADM authority for the package collection
- SYSADM authority

Syntax

```
► WLM_SET_CLIENT_INFO ( client_userid , client_wrkstnname ,  
                          NULL                          NULL  
  
                          client_applname , client_acctstr )  
                          NULL                          NULL  
◄
```

The schema is SYSPROC.

Procedure parameters

client_userid

An input argument of type VARCHAR(255) that specifies the user ID for the client. If NULL is specified, the value remains unchanged. If an empty string (") is specified, the user ID for the client is reset to the default value.

If the value specified exceeds 16 bytes, it is truncated to 16 bytes. If the value specified is less than 16 bytes, it is padded on the right with blanks to a length of 16 bytes.

client_wrkstnname

An input argument of type VARCHAR(255) that specifies the workstation name for the client. If NULL is specified, the value remains unchanged. If an empty string (") is specified, the workstation name for the client is reset to the default value.

If the value specified exceeds 18 bytes, it is truncated to 18 bytes. If the value specified is less than 18 bytes, it is padded on the right with blanks to a length of 18 bytes.

client_applname

An input argument of type VARCHAR(255) that specifies the application name for the client. If NULL is specified, the value remains unchanged. If an empty string (") is specified, the application name for the client is reset to the default value.

If the value specified exceeds 32 bytes, it is truncated to 32 bytes. If the value specified is less than 32 bytes, it is padded on the right with blanks to a length of 32 bytes.

client_acctstr

An input argument of type VARCHAR(255) that specifies the accounting string for the client. If NULL is specified, the value remains unchanged. If an empty string (") is specified, the accounting string for the client is reset to the default value.

If the requester is DB2 for z/OS, and the value that is specified exceeds 142 bytes, it is truncated to 142 bytes. Otherwise, if the value specified exceeds 200 bytes, it is truncated to 200 bytes.

Examples

Set the user ID, workstation name, application name, and accounting string for the client.

```
strcpy(user_id, "db2user");
strcpy(wkstn_name, "mywkstn");
strcpy(appl_name, "db2bp.exe");
strcpy(acct_str, "myacctstr");
iuser_id = 0;
iwkstn_name = 0;
iappl_name = 0;
iacct_str = 0;
EXEC SQL CALL SYSPROC.WLM_SET_CLIENT_INFO(:user_id:iuser_id, :wkstn_name:iwkstn_name,
                                           :appl_name:iappl_name, :acct_str:iacct_str);
```

Set the user ID to db2user for the client without setting the other client attributes.

```
strcpy(user_id, "db2user");
iuser_id = 0;
iwkstn_name = -1;
iappl_name = -1;
iacct_str = -1;
EXEC SQL CALL SYSPROC.WLM_SET_CLIENT_INFO(:user_id:iuser_id, :wkstn_name:iwkstn_name,
                                           :appl_name:iappl_name, :acct_str:iacct_str);
```

Reset the user ID for the client to blank without modifying the values of the other client attributes.

```
strcpy(user_id, "");
iuser_id = 0;
iwkstn_name = -1;
iappl_name = -1;
iacct_str = -1;
EXEC SQL CALL SYSPROC.WLM_SET_CLIENT_INFO(:user_id:iuser_id, :wkstn_name:iwkstn_name,
                                           :appl_name:iappl_name, :acct_str:iacct_str);
```

DSN_WLM_APPLENV stored procedure

This procedure provides a convenient way to define, install, and activate a new WLM application environment for DB2.

Environment

DSN_WLM_APPLENV runs in a WLM-established stored procedures address space.

Authorization

To execute the CALL DSN_WLM_APPLENV statement, the owner of the package or plan that contains the CALL statement must have one or more of the following privileges on each package that the stored procedure uses:

- The EXECUTE privilege on the DSN_WLM_APPLENV stored procedure
- Ownership of the stored procedure
- SYSADM authority

If the RACF facility class is active and a profile has been defined for the MVSADMIN.WLM.POLICY facility class, then the caller of the stored procedure needs the following access:

- ACCESS(READ) for action WLMINFO:
PERMIT MVSADMIN.WLM.POLICY CLASS(FACILITY) ID(user) ACCESS(READ)
- ACCESS(UPDATE) for action ADD, ADD_ACTIVATE, ACTIVATE:
PERMIT MVSADMIN.WLM.POLICY CLASS(FACILITY) ID(user) ACCESS(UPDATE)

Syntax

CALL DSN_WLM_APPLENV(action, policyid, wloptions, return-code, message)

```
▶▶—CALL—DSN_WLM_APPLENV—(—action—policyid—wloptions—return-code—message—)————▶▶
```

Procedure parameters

ACTION

An input parameter of type VARCHAR(20) that identifies the type of action to be performed. This is a required parameter and supports the following values:

ADD

The stored procedure will install a new WLM application environment to an existing WLM service definition without activating a service policy. The new WLM application environment will be available only on the next activation of a WLM policy.

ADD_ACTIVATE

The stored procedure will install a new WLM application environment to an existing WLM service definition and automatically activate a WLM service policy to enable the new WLM application environment.

ACTIVATE

The stored procedure will activate a WLM service policy.

WLMINFO

The stored procedure will query the existing WLM service definition and return basic information. For example, the service definition name, timestamp in the local time the service definition was installed, user ID of the service administrator that installed the service definition, name of the system on which the service definition was installed from, and number of application environment currently defined.

If the action specified is ADD or ADD_ACTIVATE, the WLMOPTIONS input parameter is also required.

POLICYID

An input parameter of type VARCHAR(8) that identifies the WLM service policy to be activated. It can be 1-8 characters long or the word ACTIVE. If the policy specified is ACTIVE, the active service policy is used. This parameter is required if the action specified is ADD_ACTIVATE or ACTIVATE, otherwise it is ignored and can be set to NULL or an empty string.

WLMOPTIONS

An input parameter of type VARCHAR(4000). This parameter is required if the action specified is ADD or ADD_ACTIVATE, otherwise it is ignored and can be

set to NULL or an empty string. The following WLM options are supported and should be separated by one or more spaces:

WLMNAME(*name*)

WLMNAME is the defined name for an application environment. It can be 1-32 characters long. It cannot begin with the letters SYS. This is a required option.

DESCRIPTION(*description*)

DESCRIPTION is a 32-character area describing the application environment. This option is not required and can be set to an empty value. For example,
DESCRIPTION()

PROCNAME(*procedure-name*)

PROCNAME defines the JCL procedure that WLM uses to start server address spaces for the application environment. It can be 1-8 characters long. This is a required option.

STARTPARM(*start-up-parameters*)

STARTPARM contains the parameters that WLM will use to start the JCL procedure. The parameters can be up to 115 characters. If the parameters include the subsystem name, the symbol &IWMSSNM can be used to cause WLM to substitute the subsystem name instead of typing the subsystem name directly. This option is useful because multiple instances of the subsystem with different names can use the application environment. For example:

```
STARTPARM(DB2SSN=&IWMSSNM,APPLENV=WLMENV1,NUMTCB=1)
```

This option is not required and can be set to an empty value.

WLMOPT(**WLM_MANAGED**|**SINGLE_SERVER**)

WLMOPT tells WLM to limit the number of server address spaces. For example, if a server address space requires exclusive use of a resource, only a single address space can exist. For DB2 routines, you can set a limit of 1 address space per system if required by the routine. Note that if there are multiple DB2 subsystems on a given system, WLM will create 1 server for each DB2 subsystem that calls the routines. A limit of 1 address space per sysplex does not apply to DB2 routines.

This option is not required and can be set to an empty value. The default is WLM_MANAGED.

Examples

Example: Returning the basic information of the WLM service definition:

```
CALL SYSPROC.DSN_WLM_APPLENV('WLMINFO', NULL, NULL, ?, ?)
```

Here is an example of the output:

```
RETURN_CODE: 0
MESSAGE: DSNT051I DSNTWLMS ACTIVE WLM SERVICE DEFINITION
          SERVICE DEFINITION NAME      WLMAMPL
          INSTALLED ON                  2010-01-25-07.11.57.764052
          INSTALLED BY                   SYSADM
          INSTALLED FROM                 LABEC130
          NUMBER OF APPL ENVIRONMENT    12
DSNT023I DSNTWLMS DISPLAY WLM INFORMATION SUCCESSFUL
```

Example: Identifying the action and WLM options:

```
CALL SYSPROC.DSN_WLM_APPLENV('ADD_ACTIVATE',
                             'ACTIVE',
                             'WLMNAME(DSNWLM_SAMPLE)
                             DESCRIPTION(DB2 SAMPLE WLM ENVIRONMENT)
                             PROCNAME(DSNWLMS)
                             STARTPARM(DB2SSN=&IWSSNM,APPLENV=' 'DSNWLM_SAMPLE' ')
                             WLMOPT(WLM_MANAGED)', ?, ?)
```

Here is an example of the output:

```
RETURN_CODE: 0
MESSAGE: DSNT023I DSNTWLMS ADD WLM APPLICATION ENVIRONMENT DSNWLM_SAMPLE SUCCESSFUL

APPLICATION ENVIRONMENT NAME : DSNWLM_SAMPLE
DESCRIPTION                   : DB2 SAMPLE WLM ENVIRONMENT
SUBSYSTEM TYPE                : DB2
PROCEDURE NAME                : DSNWLMS
START PARAMETERS              : DB2SSN=&IWSSNM,APPLENV='DSNWLM_SAMPLE'

STARTING OF SERVER ADDRESS SPACES FOR A SUBSYSTEM INSTANCE:
(x) MANAGED BY WLM
( ) LIMITED TO A SINGLE ADDRESS SPACE PER SYSTEM
( ) LIMITED TO A SINGLE ADDRESS SPACE PER SYSPLEX

DSNT023I DSNTWLMS ACTIVATE WLM POLICY WLMPLY1 SUCCESSFUL
```

DSNACICS stored procedure

The CICS transaction invocation stored procedure (DSNACICS) invokes CICS server programs.

GUPI DSNACICS gives workstation applications a way to invoke CICS server programs while using TCP/IP as their communication protocol. The workstation applications use TCP/IP and DB2 Connect to connect to a DB2 for z/OS subsystem, and then call DSNACICS to invoke the CICS server programs.

The DSNACICS input parameters require knowledge of various CICS resource definitions with which the workstation programmer might not be familiar. For this reason, DSNACICS invokes the DSNACICX user exit routine. The system programmer can write a version of DSNACICX that checks and overrides the parameters that the DSNACICS caller passes. If no user version of DSNACICX is provided, DSNACICS invokes the default version of DSNACICX, which does not modify any parameters.

Environment

DSNACICS runs in a WLM-established stored procedure address space and uses the Resource Recovery Services attachment facility to connect to DB2.

If you use CICS Transaction Server for OS/390® Version 1 Release 3 or later, you can register your CICS system as a resource manager with recoverable resource management services (RRMS). When you do that, changes to DB2 databases that are made by the program that calls DSNACICS and the CICS server program that DSNACICS invokes are in the same two-phase commit scope. This means that when the calling program performs an SQL COMMIT or ROLLBACK, DB2 and RRS inform CICS about the COMMIT or ROLLBACK.

If the CICS server program that DSNACICS invokes accesses DB2 resources, the server program runs under a separate unit of work from the original unit of work that calls the stored procedure. This means that the CICS server program might

deadlock with locks that the client program acquires.

Authorization

To execute the CALL statement, the owner of the package or plan that contains the CALL statement must have one or more of the following privileges:

- The EXECUTE privilege on stored procedure DSNACICS
- Ownership of the stored procedure
- SYSADM authority

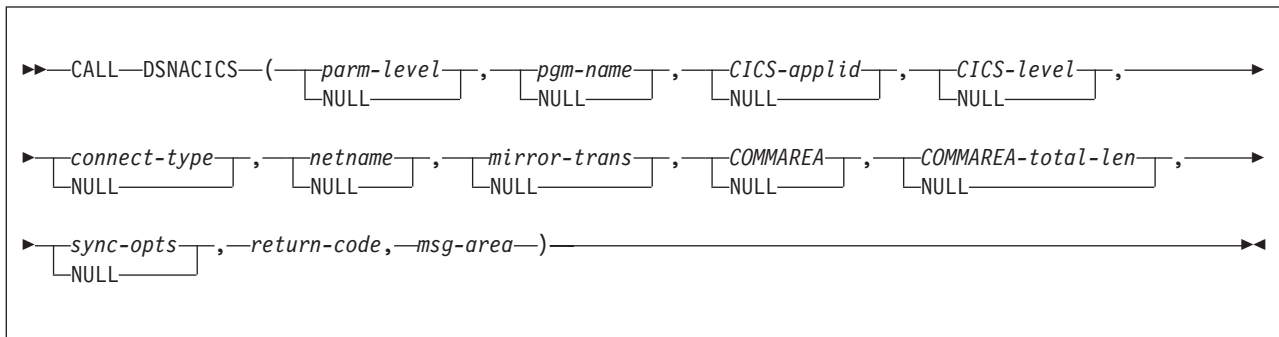
The CICS server program that DSNACICS calls runs under the same user ID as DSNACICS. That user ID depends on the SECURITY parameter that you specify when you define DSNACICS.

The DSNACICS caller also needs authorization from an external security system, such as RACF, to use CICS resources.

Syntax

The following syntax diagram shows the SQL CALL statement for invoking this stored procedure.

Because the linkage convention for DSNACICS is GENERAL WITH NULLS, if you pass parameters in host variables, you need to include a null indicator with every host variable. Null indicators for input host variables must be initialized before you execute the CALL statement.



Option descriptions

parm-level

Specifies the level of the parameter list that is supplied to the stored procedure. This is an input parameter of type INTEGER. The value must be 1.

pgm-name

Specifies the name of the CICS program that DSNACICS invokes. This is the name of the program that the CICS mirror transaction calls, not the CICS transaction name.

This is an input parameter of type CHAR(8).

CICS-applid

Specifies the applid of the CICS system to which DSNACICS connects.

This is an input parameter of type CHAR(8).

CICS-level

Specifies the level of the target CICS subsystem:

- 1 The CICS subsystem is CICS for MVS/ESA Version 4 Release 1, CICS Transaction Server for OS/390 Version 1 Release 1, or CICS Transaction Server for OS/390 Version 1 Release 2.
- 2 The CICS subsystem is CICS Transaction Server for OS/390 Version 1 Release 3 or later.

This is an input parameter of type INTEGER.

connect-type

Specifies whether the CICS connection is generic or specific. Possible values are GENERIC or SPECIFIC.

This is an input parameter of type CHAR(8).

netname

If the value of *connection-type* is SPECIFIC, specifies the name of the specific connection that is to be used. This value is ignored if the value of *connection-type* is GENERIC.

This is an input parameter of type CHAR(8).

mirror-trans

Specifies the name of the CICS mirror transaction to invoke. This mirror transaction calls the CICS server program that is specified in the *pgm-name* parameter. *mirror-trans* must be defined to the CICS server region, and the CICS resource definition for *mirror-trans* must specify DFHMIRS as the program that is associated with the transaction.

If this parameter contains blanks, DSNACICS passes a mirror transaction parameter value of null to the CICS EXCI interface. This allows an installation to override the transaction name in various CICS user-replaceable modules. If a CICS user exit routine does not specify a value for the mirror transaction name, CICS invokes CICS-supplied default mirror transaction CSMI.

This is an input parameter of type CHAR(4).

COMMAREA

Specifies the communication area (COMMAREA) that is used to pass data between the DSNACICS caller and the CICS server program that DSNACICS calls.

This is an input/output parameter of type VARCHAR(32704). In the length field of this parameter, specify the number of bytes that DSNACICS sends to the CICS server program.

commarea-total-len

Specifies the total length of the COMMAREA that the server program needs.

This is an input parameter of type INTEGER. This length must be greater than or equal to the value that you specify in the length field of the COMMAREA parameter and less than or equal to 32704. When the CICS server program completes, DSNACICS passes the server program's entire COMMAREA, which is *commarea-total-len* bytes in length, to the stored procedure caller.

sync-opts

Specifies whether the calling program controls resource recovery, using two-phase commit protocols that are supported by RRS. Possible values are:

- 1 The client program controls commit processing. The CICS server region does not perform a syncpoint when the server program returns control

to CICS. Also, the server program cannot take any explicit syncpoints. Doing so causes the server program to abnormally terminate.

- 2 The target CICS server region takes a syncpoint on successful completion of the server program. If this value is specified, the server program can take explicit syncpoints.

When CICS has been set up to be an RRS resource manager, the client application can control commit processing using SQL COMMIT requests. DB2 for z/OS ensures that CICS is notified to commit any resources that the CICS server program modifies during two-phase commit processing.

When CICS has not been set up to be an RRS resource manager, CICS forces syncpoint processing of all CICS resources at completion of the CICS server program. This commit processing is not coordinated with the commit processing of the client program.

This option is ignored when *CICS-level* is 1. This is an input parameter of type INTEGER.

return-code

Return code from the stored procedure. Possible values are:

- 0 The call completed successfully.
- 12 The request to run the CICS server program failed. The *msg-area* parameter contains messages that describe the error.

This is an output parameter of type INTEGER.

msg-area

Contains messages if an error occurs during stored procedure execution. The first messages in this area are generated by the stored procedure. Messages that are generated by CICS or the DSNACICX user exit routine might follow the first messages. The messages appear as a series of concatenated, viewable text strings.

This is an output parameter of type VARCHAR(500).

User exit routine

DSNACICS always calls user exit routine DSNACICX. You can use DSNACICX to change the values of DSNACICS input parameters before you pass those parameters to CICS. If you do not supply your own version of DSNACICX, DSNACICS calls the default DSNACICX, which modifies no values and does an immediate return to DSNACICS. The source code for the default version of DSNACICX is in member DSNASCIX in data set *prefix.SDSNSAMP*. The source code for a sample version of DSNACICX that is written in COBOL is in member DSNASCIO in data set *prefix.SDSNSAMP*.

Example

The following PL/I example shows the variable declarations and SQL CALL statement for invoking the CICS transaction that is associated with program CICSPGM1.

```
/* ***** */
/* DSNACICS PARAMETERS */
/* ***** */
DECLARE PARM_LEVEL BIN FIXED(31);
DECLARE PGM_NAME CHAR(8);
DECLARE CICS_APPLID CHAR(8);
DECLARE CICS_LEVEL BIN FIXED(31);
```

```

DECLARE CONNECT_TYPE CHAR(8);
DECLARE NETNAME      CHAR(8);
DECLARE MIRROR_TRANS CHAR(4);
DECLARE COMMAREA_TOTAL_LEN BIN FIXED(31);
DECLARE SYNC_OPTS     BIN FIXED(31);
DECLARE RET_CODE      BIN FIXED(31);
DECLARE MSG_AREA      CHAR(500) VARYING;

DECLARE1 COMMAREA BASED(P1),
      3 COMMAREA_LEN BIN FIXED(15),
      3 COMMAREA_INPUT CHAR(30),
      3 COMMAREA_OUTPUT CHAR(100);

/*****
/* INDICATOR VARIABLES FOR DSNACICS PARAMETERS */
*****/
DECLARE 1 IND_VARS,
      3 IND_PARM_LEVEL  BIN FIXED(15),
      3 IND_PGM_NAME    BIN FIXED(15),
      3 IND_CICS_APPLID BIN FIXED(15),
      3 IND_CICS_LEVEL  BIN FIXED(15),
      3 IND_CONNECT_TYPE BINFIXED(15),
      3 IND_NETNAME     BIN FIXED(15),
      3 IND_MIRROR_TRANS BIN FIXED(15),
      3 IND_COMMAREA    BIN FIXED(15),
      3 IND_COMMAREA_TOTAL_LEN BIN FIXED(15),
      3 IND_SYNC_OPTS   BIN FIXED(15),
      3 IND_RETCODE     BIN FIXED(15),
      3 IND_MSG_AREA    BIN FIXED(15);

/*****
/* LOCAL COPY OF COMMAREA */
*****/
DECLARE P1 POINTER;
DECLARE COMMAREA_STG CHAR(130) VARYING;

/*****
/* ASSIGN VALUES TO INPUT PARAMETERS PARM_LEVEL, PGM_NAME, */
/* MIRROR_TRANS, COMMAREA, COMMAREA_TOTAL_LEN, AND SYNC_OPTS. */
/* SET THE OTHER INPUT PARAMETERS TO NULL. THE DSNACICX */
/* USER EXIT MUST ASSIGN VALUES FOR THOSE PARAMETERS. */
*****/
PARM_LEVEL = 1;
IND_PARM_LEVEL = 0;

PGM_NAME = 'CICSPGM1';
IND_PGM_NAME = 0;

MIRROR_TRANS = 'MIRT';
IND_MIRROR_TRANS = 0;

P1 = ADDR(COMMAREA_STG);
COMMAREA_INPUT = 'THIS IS THE INPUT FOR CICSPGM1';
COMMAREA_OUTPUT = ' ';
COMMAREA_LEN = LENGTH(COMMAREA_INPUT);
IND_COMMAREA = 0;

COMMAREA_TOTAL_LEN = COMMAREA_LEN + LENGTH(COMMAREA_OUTPUT);
IND_COMMAREA_TOTAL_LEN = 0;

SYNC_OPTS = 1;
IND_SYNC_OPTS = 0;

IND_CICS_APPLID = -1;
IND_CICS_LEVEL = -1;
IND_CONNECT_TYPE = -1;
IND_NETNAME = -1;
/*****

```



```

/* INITIALIZE
OUTPUT PARAMETERS TO NULL. */
/*****/
IND_RETCODE = -1;
IND_MSG_AREA= -1;
/*****/
/* CALL DSNACICS TO INVOKE CICS PGM1. */
/*****/
EXEC SQL
CALL SYSPROC.DSNACICS(:PARM_LEVEL      :IND_PARM_LEVEL,
                     :PGM_NAME         :IND_PGM_NAME,
                     :CICS_APPLID      :IND_CICS_APPLID,
                     :CICS_LEVEL       :IND_CICS_LEVEL,
                     :CONNECT_TYPE     :IND_CONNECT_TYPE,
                     :NETNAME          :IND_NETNAME,
                     :MIRROR_TRANS     :IND_MIRROR_TRANS,
                     :COMMAREA_STG     :IND_COMMAREA,
                     :COMMAREA_TOTAL_LEN :IND_COMMAREA_TOTAL_LEN,
                     :SYNC_OPTS        :IND_SYNC_OPTS,
                     :RET_CODE         :IND_RET_CODE,
                     :MSG_AREA         :IND_MSG_AREA);

```

Output


DSNACICS places the return code from DSNACICS execution in the *return-code* parameter. If the value of the return code is non-zero, DSNACICS puts its own error messages and any error messages that are generated by CICS and the DSNACICX user exit routine in the *msg-area* parameter.

The *COMMAREA* parameter contains the COMMAREA for the CICS server program that DSNACICS calls. The *COMMAREA* parameter has a VARCHAR type. Therefore, if the server program puts data other than character data in the COMMAREA, that data can become corrupted by code page translation as it is passed to the caller. To avoid code page translation, you can change the COMMAREA parameter in the CREATE PROCEDURE statement for DSNACICS to VARCHAR(32704) FOR BIT DATA. However, if you do so, the client program might need to do code page translation on any character data in the COMMAREA to make it readable.

Restrictions

Because DSNACICS uses the distributed program link (DPL) function to invoke CICS server programs, server programs that you invoke through DSNACICS can contain only the CICS API commands that the DPL function supports.

Debugging

If you receive errors when you call DSNACICS, ask your system administrator to add a DSNDUMP DD statement in the startup procedure for the address space in which DSNACICS runs. The DSNDUMP DD statement causes DB2 to generate an SVC dump whenever DSNACICS issues an error message. 

Related information:

 Accessing CICS systems through stored procedure DSNACICS (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

 The API commands (CICS Transaction Server for z/OS)

The DSNACICX user exit routine

Use DSNACICX to change the values of DSNACICS input parameters before you pass those parameters to CICS.

GUPI

General considerations

The DSNACICX exit routine must follow these rules:

- It can be written in assembler, COBOL, PL/I, or C.
- It must follow the Language Environment calling linkage when the caller is an assembler language program.
- The load module for DSNACICX must reside in an authorized program library that is in the STEPLIB concatenation of the stored procedure address space startup procedure.

You can replace the default DSNACICX in the *prefix.SDSNLOAD*, library, or you can put the DSNACICX load module in a library that is ahead of *prefix.SDSNLOAD* in the STEPLIB concatenation. It is recommended that you put DSNACICX in the *prefix.SDSNEXIT* library. Sample installation job DSNTIJEX contains JCL for assembling and link-editing the sample source code for DSNACICX into *prefix.SDSNEXIT*. You need to modify the JCL for the libraries and the compiler that you are using.

- The load module must be named DSNACICX.
- The exit routine must save and restore the caller's registers. Only the contents of register 15 can be modified.
- It must be written to be reentrant and link-edited as reentrant.
- It must be written and link-edited to execute as AMODE(31),RMODE(ANY).
- DSNACICX can contain SQL statements. However, if it does, you need to change the DSNACICS procedure definition to reflect the appropriate SQL access level for the types of SQL statements that you use in the user exit routine.

Specifying the exit routine

DSNACICS always calls an exit routine named DSNACICX. DSNACICS calls your DSNACICX exit routine if it finds it before the default DSNACICX exit routine. Otherwise, it calls the default DSNACICX exit routine.

When the exit routine is taken

The DSNACICX exit routine is taken whenever DSNACICS is called. The exit routine is taken before DSNACICS invokes the CICS server program.

Loading a new version of the exit routine

DB2 loads DSNACICX only once, when DSNACICS is first invoked. If you change DSNACICX, you can load the new version by quiescing and then resuming the WLM application environment for the stored procedure address space in which DSNACICS runs:

```
VARY WLM,APPLENV=DSNACICS-applenv-name,QUIESCE VARY  
WLM,APPLENV=DSNACICS-applenv-name,RESUME
```

Parameter list

At invocation, registers are set as described in the following table

Table 134. Registers at invocation of DSNACICX

Register	Contains
1	Address of pointer to the exit parameter list (XPL).
13	Address of the register save area.
14	Return address.
15	Address of entry point of exit routine.

The following table shows the contents of the DSNACICX exit parameter list, XPL. Member DSNDXPL in data set *prefix.SDSNMACS* contains an assembler language mapping macro for XPL. Sample exit routine DSNASCIO in data set *prefix.SDSNSAMP* includes a COBOL mapping macro for XPL.

Table 135. Contents of the XPL exit parameter list

Name	Hex offset	Data type	Description	Corresponding DSNACICS parameter
XPL_EYEC	0	Character, 4 bytes	Eye-catcher: 'XPL '	
XPL_LEN	4	Character, 4 bytes	Length of the exit parameter list	
XPL_LEVEL	8	4-byte integer	Level of the parameter list	<i>parm-level</i>
XPL_PGMNAME	C	Character, 8 bytes	Name of the CICS server program	<i>pgm-name</i>
XPL_CICSAPPLID	14	Character, 8 bytes	CICS VTAM® applid	<i>CICS-applid</i>
XPL_CICSLEVEL	1C	4-byte integer	Level of CICS code	<i>CICS-level</i>
XPL_CONNECTTYPE	20	Character, 8 bytes	Specific or generic connection to CICS	<i>connect-type</i>
XPL_NETNAME	28	Character, 8 bytes	Name of the specific connection to CICS	<i>netname</i>
XPL_MIRRORTRAN	30	Character, 8 bytes	Name of the mirror transaction that invokes the CICS server program	<i>mirror-trans</i>
XPL_COMMAREAPTR	38	Address, 4 bytes	Address of the COMMAREA	¹
XPL_COMMINLEN	3C	4-byte integer	Length of the COMMAREA that is passed to the server program	²
XPL_COMMTOTLEN	40	4-byte integer	Total length of the COMMAREA that is returned to the caller	<i>commarea-total-len</i>
XPL_SYNCOPTS	44	4-byte integer	Syncpoint control option	<i>sync-opts</i>

Table 135. Contents of the XPL exit parameter list (continued)

Name	Hex offset	Data type	Description	Corresponding DSNACICS parameter
XPL_RETCODE	48	4-byte integer	Return code from the exit routine	<i>return-code</i>
XPL_MSGLLEN	4C	4-byte integer	Length of the output message area	<i>return-code</i>
XPL_MSGAREA	50	Character, 256 bytes	Output message area	<i>msg-area</i> ³

Notes:

1. The area that this field points to is specified by DSNACICS parameter *COMMAREA*. This area does not include the length bytes.
2. This is the same value that the DSNACICS caller specifies in the length bytes of the *COMMAREA* parameter.
3. Although the total length of *msg-area* is 500 bytes, DSNACICX can use only 256 bytes of that area.



DSNAIMS stored procedure

DSNAIMS is a stored procedure that allows DB2 applications to invoke IMS transactions and commands easily, without maintaining their own connections to IMS.



DSNAIMS uses the IMS Open Transaction Manager Access (OTMA) API to connect to IMS and execute the transactions.

Environment

DSNAIMS runs in a WLM-established stored procedures address space. DSNAIMS requires DB2 with RRSF enabled and IMS version 7 or later with OTMA Callable Interface enabled.

To use a two-phase commit process, you must have IMS Version 8 with UQ70789 or later.

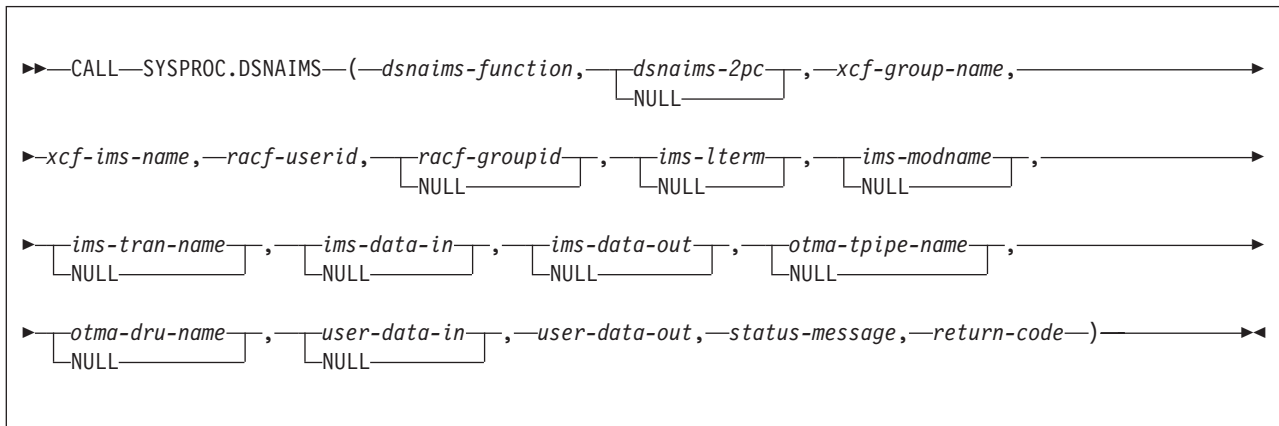
Authorization

To set up and run DSNAIMS, you must be authorized the perform the following steps:

1. Use the job DSNTIJIM to issue the CREATE PROCEDURE statement for DSNAIMS and to grant the execution of DSNAIMS to PUBLIC. DSNTIJIM is provided in the SDSNSAMP data set. You need to customize DSNTIJIM to fit the parameters of your system.
2. Ensure that OTMA C/I is initialized.

Syntax

The following syntax diagram shows the SQL CALL statement for invoking this stored procedure:



Option descriptions

dsnaims-function

A string that indicates whether the transaction is send-only, receive-only, or send-and-receive. Possible values are:

SENDRECV

Sends and receives IMS data. SENDRECV invokes an IMS transaction or command and returns the result to the caller. The transaction can be an IMS full function or a fast path. SENDRECV does not support multiple iterations of a conversational transaction

SEND Sends IMS data. SEND invokes an IMS transaction or command, but does not receive IMS data. If result data exists, it can be retrieved with the RECEIVE function. A send-only transaction cannot be an IMS fast path transaction or a conversational transaction.

RECEIVE

Receives IMS data. The data can be the result of a transaction or command initiated by the SEND function or an unsolicited output message from an IMS application. The RECEIVE function does not initiate an IMS transaction or command.

dsnaims-2pc

Specifies whether to use a two-phase commit process to perform the transaction syncpoint service. Possible values are Y or N. For N, commits and rollbacks that are issued by the IMS transaction do not affect commit and rollback processing in the DB2 application that invokes DSNAIMS. Furthermore, IMS resources are not affected by commits and rollbacks that are issued by the calling DB2 application. If you specify Y, you must also specify SENDRECV. To use a two-phase commit process, you must set the IMS control region parameter (RRS) to Y.

This parameter is optional. The default is N.

xcf-group-name

Specifies the XCF group name that the IMS OTMA joins. You can obtain this name by viewing the GRNAME parameter in IMS PROCLIB member DFSPBxxx or by using the IMS command /DISPLAY OTMA.

xcf-ims-name

Specifies the XCF member name that IMS uses for the XCF group. If IMS is not using the XRF or RSR feature, you can obtain the XCF member name from the OTMANM parameter in IMS PROCLIB member DFSPBxxx. If IMS is using the XRF or RSR feature, you can obtain the XCF member name from the USERVAR parameter in IMS PROCLIB member DFSPBxxx.

racf-userid

Specifies the RACF user ID that is used for IMS to perform the transaction or command authorization checking. This parameter is required if DSNAIMS is running APF-authorized. If DSNAIMS is running unauthorized, this parameter is ignored and the EXTERNAL SECURITY setting for the DSNAIMS stored procedure definition determines the user ID that is used by IMS.

racf-groupid

Specifies the RACF group ID that is used for IMS to perform the transaction or command authorization checking. This field is used for stored procedures that are APF-authorized. It is ignored for other stored procedures.

ims-lterm

Specifies an IMS LTERM name that is used to override the LTERM name in the I/O program communication block of the IMS application program.

This field is used as an input and an output field:

- For SENDRECV, the value is sent to IMS on input and can be updated by IMS on output.
- For SEND, the parameter is IN only.
- For RECEIVE, the parameter is OUT only.

An empty or NULL value tells IMS to ignore the parameter.

ims-modname

Specifies the formatting map name that is used by the server to map output data streams, such as 3270 streams. Although this invocation does not have IMS MFS support, the input MODNAME can be used as the map name to define the output data stream. This name is an 8-byte message output descriptor name that is placed in the I/O program communication block. When the message is inserted, IMS places this name in the message prefix with the map name in the program communication block of the IMS application program.

For SENDRECV, the value is sent to IMS on input, and can be updated on output. For SEND, the parameter is IN only. For RECEIVE it is OUT only. IMS ignores the parameter when it is an empty or NULL value.

ims-tran-name

Specifies the name of an IMS transaction or command that is sent to IMS. If the IMS command is longer than eight characters, specify the first eight characters (including the "/" of the command). Specify the remaining characters of the command in the *ims-tran-name* parameter. If you use an empty or NULL value, you must specify the full transaction name or command in the *ims-data-in* parameter.

ims-data-in

Specifies the data that is sent to IMS. This parameter is required in each of the following cases:

- Input data is required for IMS
- No transaction name or command is passed in *ims-tran-name*
- The command is longer than eight characters

This parameter is ignored when for RECEIVE functions.

ims-data-out

Data returned after successful completion of the transaction. This parameter is required for SENDRECV and RECEIVE functions. The parameter is ignored for SEND functions.

The length of *ims-data-out* is 32,000 bytes. If the data that is returned from IMS is greater than the length of *ims-data-out*, the data will be truncated.

otma-tpipe-name

Specifies an 8-byte user-defined communication session name that IMS uses for the input and output data for the transaction or the command in a SEND or a RECEIVE function. If the *otma_tpipe_name* parameter is used for a SEND function to generate an IMS output message, the same *otma_pipe_name* must be used to retrieve output data for the subsequent RECEIVE function.

otma-dru-name

Specifies the name of an IMS user-defined exit routine, OTMA destination resolution user exit routine, if it is used. This IMS exit routine can format part of the output prefix and can determine the output destination for an IMS ALT_PCB output. If an empty or null value is passed, IMS ignores this parameter.

user-data-in

This optional parameter contains any data that is to be included in the IMS message prefix, so that the data can be accessed by IMS OTMA user exit routines (DFSYIOE0 and DFSYDRU0) and can be tracked by IMS log records. IMS applications that run in dependent regions do not access this data. The specified user data is not included in the output message prefix. You can use this parameter to store input and output correlator tokens or other information. This parameter is ignored for RECEIEVE functions.

user-data-out

On output, this field contains the *user-data-in* in the IMS output prefix. IMS user exit routines (DFSYIOE0 and DFSYDRU0) can also create *user-data-out* for SENDRECV and RECEIVE functions. The parameter is not updated for SEND functions.

The length of *user-data-out* is 1,022 bytes. If the data that is returned from IMS is greater than the length of *user-data-out*, the data will be truncated.

status-message

Indicates any error message that is returned from the transaction or command, OTMA, RRS, or DSNAIMS.

return-code

Indicates the return code that is returned for the transaction or command, OTMA, RRS, or DSNAIMS.

Examples

The following examples show how to call DSNAIMS.

Example 1: Sample parameters for executing an IMS command:

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM",  
                    "IMSCCLNM", "", "", "", "", "", "",  
                    "/LOG Hello World.", ims_data_out, "", "", "",  
                    user_out, error_message, rc)
```

Example 2: Sample parameters for executing an IMS IVTNO transaction:

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM",
                    "IMSCCLNM", "", "", "", "", "",
                    "IVTNO    DISPLAY LAST1    ", ims_data_out
                    "", "", "", user_out, error_message, rc)
```

Example 3: Sample parameters for send-only IMS transaction:

```
CALL SYSPROC.DSNAIMS("SEND", "N", "IMS7GRP", "IMS7TMEM",
                    "IMSCCLNM", "", "", "", "", "",
                    "IVTNO    DISPLAY LAST1    ", ims_data_out,
                    "DSNAPIPE", "", "", user_out, error_message, rc)
```

Example 4: Sample parameters for receive-only IMS transaction:

```
CALL SYSPROC.DSNAIMS("RECEIVE", "N", "IMS7GRP", "IMS7TMEM",
                    "IMSCCLNM", "", "", "", "", "",
                    "IVTNO    DISPLAY LAST1    ", ims_data_out,
                    "DSNAPIPE", "", "", user_out, error_message, rc)
```

Connecting to multiple IMS subsystems with DSNAIMS

By default DSNAIMS connects to only one IMS subsystem at a time. The first request to DSNAIMS determines to which IMS subsystem the stored procedure connects. DSNAIMS attempts to reconnect to IMS only in the following cases:

- IMS is restarted and the saved connection is no longer valid
- WLM loads another DSNAIMS task

To connect to multiple IMS subsystems simultaneously, perform the following steps:

1. Make a copy of the DB2-supplied job DSNTIJIM and customize it to your environment.
2. Change the procedure name from SYSPROC.DSNAIMS to another name, such as DSNAIMSB.
3. Do not change the EXTERNAL NAME option. Leave it as DSNAIMS.
4. Run the new job to create a second instance of the stored procedure.
5. To ensure that you connect to the intended IMS target, consistently use the XFC group and member names that you associate with each stored procedure instance. For example:

```
CALL SYSPROC.DSNAIMS("SENDRECV", "N", "IMS7GRP", "IMS7TMEM", ...)
CALL SYSPROC.DSNAIMSB("SENDRECV", "N", "IMS8GRP", "IMS8TMEM", ...)
```



Related concepts:

OTMA C/I initialization

Related information:

Accessing IMS databases from DB2 stored procedures (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

DSNAIMS2 stored procedure

DSNAIMS2 is a stored procedure that allows DB2 applications to invoke IMS transactions and commands easily, without maintaining their own connections to IMS. DSNAIMS2 includes multi-segment input support for IMS transactions.

GUIP DSNAIMS2 uses the IMS Open Transaction Manager Access (OTMA) API to connect to IMS and execute the transactions.

When you define the DSNAIMS2 stored procedure to your DB2 subsystem, you can use the name DSNAIMS in your application if you prefer. Customize DSNTIJI2 to define the stored procedure to your DB2 subsystem as DSNAIMS; however, the EXTERNAL NAME option must still be DSNAIMS2.

Environment

DSNAIMS2 runs in a WLM-established stored procedures address space. DSNAIMS2 requires DB2 with RRSF enabled and IMS version 7 or later with OTMA Callable Interface enabled.

To use a two-phase commit process, you must have IMS Version 8 with UQ70789 or later.

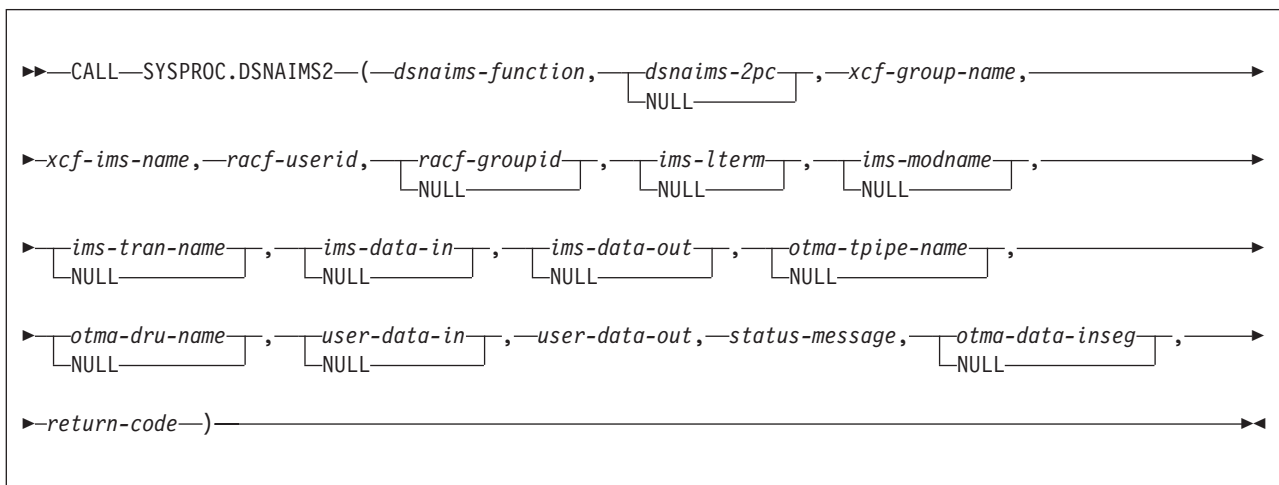
Authorization

To set up and run DSNAIMS2, you must be authorized to perform the following steps:

1. Use the job DSNTIJI2 to issue the CREATE PROCEDURE statement for DSNAIMS2 and to grant the execution of DSNAIMS2 to PUBLIC. DSNTIJI2 is provided in the SDSNSAMP data set. You need to customize DSNTIJI2 to fit the parameters of your system.
2. Ensure that OTMA C/I is initialized.

Syntax

The following syntax diagram shows the SQL CALL statement for invoking this stored procedure:



Option descriptions

dsnaims-function

A string that indicates whether the transaction is send-only, receive-only, or send-and-receive. Possible values are:

SENDRECV

Sends and receives IMS data. SENDRECV invokes an IMS transaction or command and returns the result to the caller. The transaction can be an IMS full function or a fast path. SENDRECV does not support multiple iterations of a conversational transaction

SEND Sends IMS data. SEND invokes an IMS transaction or command, but does not receive IMS data. If result data exists, it can be retrieved with the RECEIVE function. A send-only transaction cannot be an IMS fast path transaction or a conversations transaction.

RECEIVE

Receives IMS data. The data can be the result of a transaction or command initiated by the SEND function or an unsolicited output message from an IMS application. The RECEIVE function does not initiate an IMS transaction or command.

dsnaims-2pc

Specifies whether to use a two-phase commit process to perform the transaction syncpoint service. Possible values are Y or N. For N, commits and rollbacks that are issued by the IMS transaction do not affect commit and rollback processing in the DB2 application that invokes DSNAIMS2. Furthermore, IMS resources are not affected by commits and rollbacks that are issued by the calling DB2 application. If you specify Y, you must also specify SENDRECV. To use a two-phase commit process, you must set the IMS control region parameter (RRS) to Y.

This parameter is optional. The default is N.

xcf-group-name

Specifies the XCF group name that the IMS OTMA joins. You can obtain this name by viewing the GRNAME parameter in IMS PROCLIB member DFSPBxxx or by using the IMS command /DISPLAY OTMA.

xcf-ims-name

Specifies the XCF member name that IMS uses for the XCF group. If IMS is not using the XRF or RSR feature, you can obtain the XCF member name from the OTMANM parameter in IMS PROCLIB member DFSPBxxx. If IMS is using the XRF or RSR feature, you can obtain the XCF member name from the USERVAR parameter in IMS PROCLIB member DFSPBxxx.

racf-userid

Specifies the RACF user ID that is used for IMS to perform the transaction or command authorization checking. This parameter is required if DSNAIMS2 is running APF-authorized. If DSNAIMS2 is running unauthorized, this parameter is ignored and the EXTERNAL SECURITY setting for the DSNAIMS2 stored procedure definition determines the user ID that is used by IMS.

racf-groupid

Specifies the RACF group ID that is used for IMS to perform the transaction or command authorization checking. This field is used for stored procedures that are APF-authorized. It is ignored for other stored procedures.

ims-lterm

Specifies an IMS LTERM name that is used to override the LTERM name in the I/O program communication block of the IMS application program.

This field is used as an input and an output field:

- For SENDRECV, the value is sent to IMS on input and can be updated by IMS on output.

- For SEND, the parameter is IN only.
- For RECEIVE, the parameter is OUT only.

An empty or NULL value tells IMS to ignore the parameter.

ims-modname

Specifies the formatting map name that is used by the server to map output data streams, such as 3270 streams. Although this invocation does not have IMS MFS support, the input MODNAME can be used as the map name to define the output data stream. This name is an 8-byte message output descriptor name that is placed in the I/O program communication block. When the message is inserted, IMS places this name in the message prefix with the map name in the program communication block of the IMS application program.

For SENDRECV, the value is sent to IMS on input, and can be updated on output. For SEND, the parameter is IN only. For RECEIVE it is OUT only. IMS ignores the parameter when it is an empty or NULL value.

ims-tran-name

Specifies the name of an IMS transaction or command that is sent to IMS. If the IMS command is longer than eight characters, specify the first eight characters (including the "/" of the command). Specify the remaining characters of the command in the *ims-tran-name* parameter. If you use an empty or NULL value, you must specify the full transaction name or command in the *ims-data-in* parameter.

ims-data-in

Specifies the data that is sent to IMS. This parameter is required in each of the following cases:

- Input data is required for IMS
- No transaction name or command is passed in *ims-tran-name*
- The command is longer than eight characters

This parameter is ignored when for RECEIVE functions.

ims-data-out

Data returned after successful completion of the transaction. This parameter is required for SENDRECV and RECEIVE functions. The parameter is ignored for SEND functions.

The length of *ims-data-out* is 32,000 bytes. If the data that is returned from IMS is greater than the length of *ims-data-out*, the data will be truncated.

otma-tpipe-name

Specifies an 8-byte user-defined communication session name that IMS uses for the input and output data for the transaction or the command in a SEND or a RECEIVE function. If the *otma_tpipe_name* parameter is used for a SEND function to generate an IMS output message, the same *otma_pipe_name* must be used to retrieve output data for the subsequent RECEIVE function.

otma-dru-name

Specifies the name of an IMS user-defined exit routine, OTMA destination resolution user exit routine, if it is used. This IMS exit routine can format part of the output prefix and can determine the output destination for an IMS ALT_PCB output. If an empty or null value is passed, IMS ignores this parameter.

user-data-in

This optional parameter contains any data that is to be included in the IMS

message prefix, so that the data can be accessed by IMS OTMA user exit routines (DFSYIOE0 and DFSYDRU0) and can be tracked by IMS log records. IMS applications that run in dependent regions do not access this data. The specified user data is not included in the output message prefix. You can use this parameter to store input and output correlator tokens or other information. This parameter is ignored for RECEIIVE functions.

user-data-out

On output, this field contains the *user-data-in* in the IMS output prefix. IMS user exit routines (DFSYIOE0 and DFSYDRU0) can also create *user-data-out* for SENDRECV and RECEIVE functions. The parameter is not updated for SEND functions.

The length of *user-data-out* is 1,022 bytes. If the data that is returned from IMS is greater than the length of *user-data-out*, the data will be truncated.

status-message

Indicates any error message that is returned from the transaction or command, OTMA, RRS, or DSNAIMS2.

otma-data-inseg

Specifies the number of segments followed by the lengths of the segments to be sent to IMS. All values should be separated by semicolons. This field is required to send multi-segment input to IMS. For single-segment transactions and commands, set the field to NULL, "0" or "0;".

return-code

Indicates the return code that is returned for the transaction or command, OTMA, RRS, or DSNAIMS2.

Examples

The following examples show how to call DSNAIMS2.

Example 1: Sample parameters for executing a multi-segment IMS transaction:

```
CALL SYSPROC.DSNAIMS2("SEND","N","IMS7GRP","IMS7TMEM",
    "IMSCLNM","","","","","","",
    "PART 1ST SEGMENT FROM CI 2ND SEGMENT FROM CI ",
    ims_data_out,"","","",user_out, error_message,
    "2;25;20",rc)
```

Example 2: Sample parameters for executing a single-segment IMS IVTNO transaction:

```
CALL SYSPROC.DSNAIMS2("SEND","N","IMS7GRP","IMS7TMEM",
    "IMSCLNM","","","","","","IVTNO",
    "DISPLAY LAST1",ims_data_out,"","","",
    user_out, error_message,NULL,rc)
```

Connecting to multiple IMS subsystems with DSNAIMS2

By default DSNAIMS2 connects to only one IMS subsystem at a time. The first request to DSNAIMS2 determines to which IMS subsystem the stored procedure connects. DSNAIMS2 attempts to reconnect to IMS only in the following cases:

- IMS is restarted and the saved connection is no longer valid
- WLM loads another DSNAIMS2 task

To connect to multiple IMS subsystems simultaneously, perform the following steps:

1. Make a copy of the DB2-supplied job DSNTIJI2 and customize it to your environment.
2. Change the procedure name from SYSPROC.DSNAIMS2 to another name, such as DSNAIMS2B.
3. Do not change the EXTERNAL NAME option. Leave it as DSNAIMS2.
4. Change the name of the stored procedure in the grant statement in job DSNTIJI2.
5. Run the new job to create a second instance of the stored procedure.
6. To ensure that you connect to the intended IMS target, consistently use the XFC group and member names that you associate with each stored procedure instance. For example:

```
CALL SYSPROC.DSNAIMS2("SENDRECV", "N", "IMS7GRP", "IMS7TMEM", ...)
CALL SYSPROC.DSNAIMS2B("SENDRECV", "N", "IMS8GRP", "IMS8TMEM", ...)
```

GUIP

Related concepts:

 [OTMA C/I initialization](#)

Related information:

 [Accessing IMS databases from DB2 stored procedures \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

DSNACCOR stored procedure (deprecated)

The DB2 real-time statistics stored procedure (DSNACCOR) is a sample stored procedure that makes recommendations to help you maintain your DB2 databases. The DSNACCOX stored procedure replaces the DSNACCOR stored procedure, which is deprecated, and provides improved recommendations. You can continue to use the DSNACCOR stored procedure. However, DSNACCOR is not enhanced with new fields, improved formulas, and other enhancements in found in the DSNACCOX stored procedure, including the option to select the formula that is used for making recommendations.

PSPI

In particular, DSNACCOR performs the following actions:

- Recommends when you should reorganize, image copy, or update statistics for table spaces or index spaces
- Indicates when a data set has exceeded a specified threshold for the number of extents that it occupies.
- Indicates whether objects are in a restricted state

DSNACCOR uses data from the SYSIBM.SYSTABLESPACESTATS and SYSIBM.SYSSYSINDEXSPACESTATS real-time statistics tables to make its recommendations. DSNACCOR provides its recommendations in a result set.

DSNACCOR uses the set of criteria that are shown in “DSNACCOR formulas for recommending actions” on page 852 to evaluate table spaces and index spaces. By default, DSNACCOR evaluates **all** table spaces and index spaces in the subsystem that have entries in the real-time statistics tables. However, you can override this default through input parameters.

Important information about DSNACCOR recommendations:

- DSNACCOR makes recommendations based on general formulas that require input from the user about the maintenance policies for a subsystem. These recommendations might not be accurate for every installation.
- If the real-time statistics tables contain information for only a small percentage of your DB2 subsystem, the recommendations that DSNACCOR makes might not be accurate for the entire subsystem.
- Before you perform any action that DSNACCOR recommends, ensure that the object for which DSNACCOR makes the recommendation is available, and that the recommended action can be performed on that object. For example, before you can perform an image copy on an index, the index must have the COPY YES attribute.

Environment

DSNACCOR must run in a WLM-established stored procedure address space. The DSNWLM_GENERAL core WLM environment is a suitable environment for this stored procedure.

DSNACCOR is installed and configured by installation job DSNTIJRT, which binds the package for DSNACCOR with isolation UR to avoid lock contention.

Authorization required

To execute the CALL DSNACCOR statement, the owner of the package or plan that contains the CALL statement must have one or more of the following privileges on each package that the stored procedure uses:

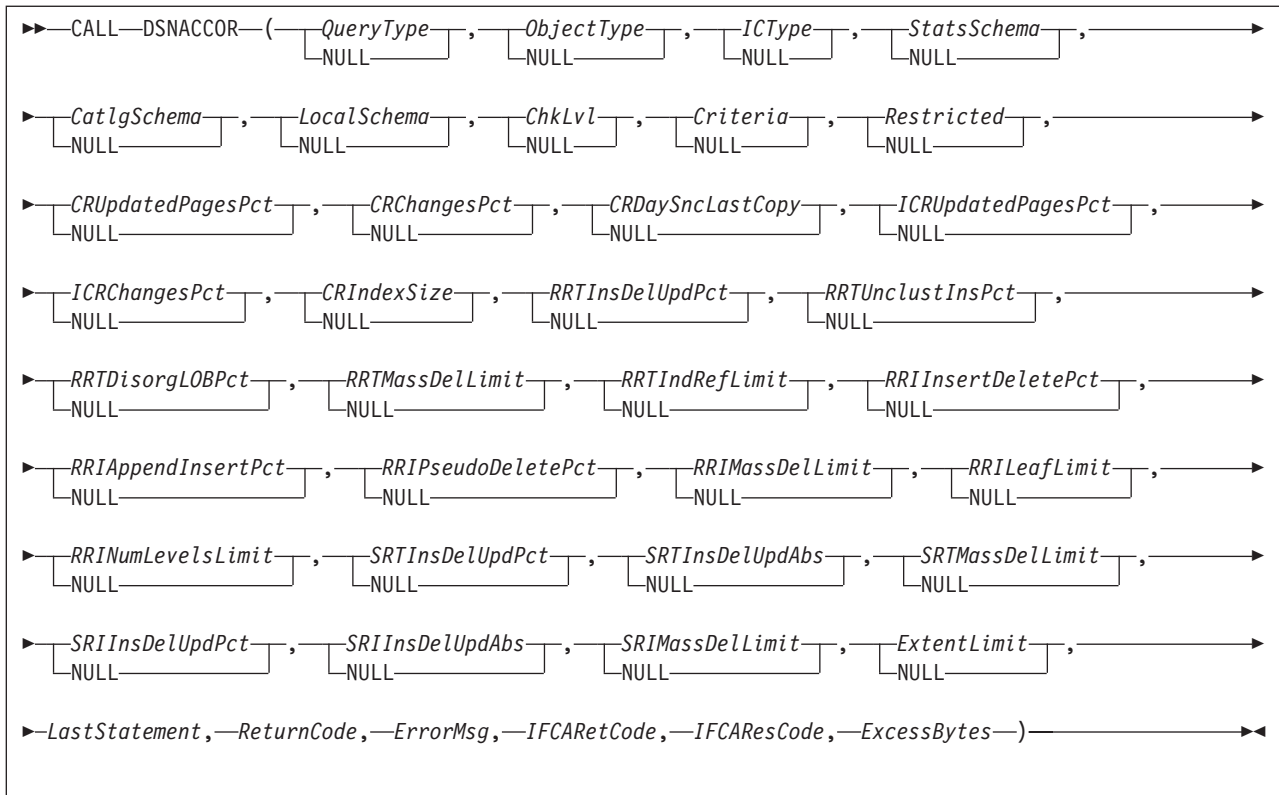
- The EXECUTE privilege on the package for DSNACCOR
- Ownership of the package
- PACKADM authority for the package collection
- SYSADM authority

The owner of the package or plan that contains the CALL statement must also have:

- SELECT authority on the real-time statistics tables
- Select authority on catalog tables
- The DISPLAY system privilege

Syntax diagram

The following syntax diagram shows the CALL statement for invoking DSNACCOR. Because the linkage convention for DSNACCOR is GENERAL WITH NULLS, if you pass parameters in host variables, you need to include a null indicator with every host variable. Null indicators for input host variables must be initialized before you execute the CALL statement.



Option descriptions

In the following option descriptions, the default value for an input parameter is the value that DSNACCOR uses if you specify a null value.

QueryType

Specifies the types of actions that DSNACCOR recommends. This field contains one or more of the following values. Each value is enclosed in single quotation marks and separated from other values by a space.

ALL Makes recommendations for all of the following actions.

COPY Makes a recommendation on whether to perform an image copy.

RUNSTATS

Makes a recommendation on whether to perform RUNSTATS.

REORG

Makes a recommendation on whether to perform REORG. Choosing this value causes DSNACCOR to process the EXTENTS value also.

EXTENTS

Indicates when data sets have exceeded a user-specified extents limit.

RESTRICT

Indicates which objects are in a restricted state.

QueryType is an input parameter of type VARCHAR(40). The default is **ALL**.

ObjectType

Specifies the types of objects for which DSNACCOR recommends actions:

ALL Table spaces and index spaces.

TS Table spaces only.

IX Index spaces only.

ObjectType is an input parameter of type VARCHAR(3). The default is **ALL**.

ICType

Specifies the types of image copies for which DSNACCOR is to make recommendations:

F Full image copy.

I Incremental image copy. This value is valid for table spaces only.

B Full image copy or incremental image copy.

ICType is an input parameter of type VARCHAR(1). The default is **B**.

StatsSchema

Specifies the qualifier for the real-time statistics table names. *StatsSchema* is an input parameter of type VARCHAR(128). The default is **SYSIBM**.

CatlgSchema

Specifies the qualifier for DB2 catalog table names. *CatlgSchema* is an input parameter of type VARCHAR(128). The default is **SYSIBM**.

LocalSchema

Specifies the qualifier for the names of tables that DSNACCOR creates. *LocalSchema* is an input parameter of type VARCHAR(128). The default is **DSNACC**.

ChkLvl

Specifies the types of checking that DSNACCOR performs, and indicates whether to include objects that fail those checks in the DSNACCOR recommendations result set. This value is the sum of any combination of the following values:

0 DSNACCOR performs none of the following actions.

1 For objects that are listed in the recommendations result set, check the SYSTABLESPACE or SYSINDEXES catalog tables to ensure that those objects have not been deleted. If value 16 is **not** also chosen, exclude rows for the deleted objects from the recommendations result set.

DSNACCOR excludes objects from the recommendations result set if those objects are not in the SYSTABLESPACE or SYSINDEXES catalog tables.

When this setting is specified, DSNACCOR does not use EXTENTS>*ExtentLimit* to determine whether a LOB table space should be reorganized.

2 For index spaces that are listed in the recommendations result set, check the SYSTABLES, SYSTABLESPACE, and SYSINDEXES catalog tables to determine the name of the table space that is associated with each index space.

Choosing this value causes DSNACCOR to also check for rows in the recommendations result set for objects that have been deleted but have entries in the real-time statistics tables (value 1). This means that if value 16 is **not** also chosen, rows for deleted objects are excluded from the recommendations result set.

4 Check whether rows that are in the DSNACCOR recommendations result set refer to objects that are in the exception table. For recommendations result set rows that have corresponding exception

table rows, copy the contents of the QUERYTYPE column of the exception table to the INEXCEPTTABLE column of the recommendations result set.

- 8 Check whether objects that have rows in the recommendations result set are restricted. Indicate the restricted status in the OBJECTSTATUS column of the result set. A row is added to the result set for each object that has a restricted state, even if a row for the same object is already included in the result set because utility operations are recommended. So, the result set might contain duplicate rows for the same object when you specify this option.
- 16 For objects that are listed in the recommendations result set, check the SYSTABLESPACE or SYSINDEXES catalog tables to ensure that those objects have not been deleted (value 1). In result set rows for deleted objects, specify the word ORPHANED in the OBJECTSTATUS column.
- 32 Exclude rows from the DSNACCOR recommendations result set for index spaces for which the related table spaces have been recommended for REORG. Choosing this value causes DSNACCOR to perform the actions for values 1 and 2.
- 64 For index spaces that are listed in the DSNACCOR recommendations result set, check whether the related table spaces are listed in the exception table. For recommendations result set rows that have corresponding exception table rows, copy the contents of the QUERYTYPE column of the exception table to the INEXCEPTTABLE column of the recommendations result set.

ChkLvl is an input parameter of type INTEGER. The default is 7 (values 1+2+4).

Criteria

Narrows the set of objects for which DSNACCOR makes recommendations. This value is the search condition of an SQL WHERE clause. *Criteria* is an input parameter of type VARCHAR(4096). The default is that DSNACCOR makes recommendations for all table spaces and index spaces in the subsystem. The search condition can use any column in the result set and wildcards are allowed.

Restricted

A parameter that is reserved for future use. Specify the null value for this parameter. *Restricted* is an input parameter of type VARCHAR(80).

CRUpdatedPagesPct

Specifies a criterion for recommending a full image copy on a table space or index space. If the following condition is true for a table space, DSNACCOR recommends an image copy:

The total number of distinct updated pages, divided by the total number of preformatted pages (expressed as a percentage) is greater than *CRUpdatedPagesPct*.

See item 2 in Figure 42 on page 852. If both of the following conditions are true for an index space, DSNACCOR recommends an image copy:

- The total number of distinct updated pages, divided by the total number of preformatted pages (expressed as a percentage) is greater than *CRUpdatedPagesPct*.
- The number of active pages in the index space or partition is greater than *CRIndexSize*. See items 2 and 3 in Figure 43 on page 853.

CRUpdatedPagesPct is an input parameter of type INTEGER. The default is 20.

CRChangesPct

Specifies a criterion for recommending a full image copy on a table space or index space. If the following condition is true for a table space, DSNACCOR recommends an image copy:

The total number of insert, update, and delete operations since the last image copy, divided by the total number of rows or LOBs in a table space or partition (expressed as a percentage) is greater than *CRChangesPct*.

See item 3 in Figure 42 on page 852. If both of the following conditions are true for an index table space, DSNACCOR recommends an image copy:

- The total number of insert and delete operations since the last image copy, divided by the total number of entries in the index space or partition (expressed as a percentage) is greater than *CRChangesPct*.
- The number of active pages in the index space or partition is greater than *CRIndexSize*.

See items 2 and 4 in Figure 43 on page 853. *CRChangesPct* is an input parameter of type INTEGER. The default is 10.

CRDaySncLastCopy

Specifies a criterion for recommending a full image copy on a table space or index space. If the number of days since the last image copy is greater than this value, DSNACCOR recommends an image copy. (See item 1 in Figure 42 on page 852 and item 1 in Figure 43 on page 853.) *CRDaySncLastCopy* is an input parameter of type INTEGER. The default is 7.

ICRUpdatedPagesPct

Specifies a criterion for recommending an incremental image copy on a table space. If the following condition is true, DSNACCOR recommends an incremental image copy:

The number of distinct pages that were updated since the last image copy, divided by the total number of active pages in the table space or partition (expressed as a percentage) is greater than *ICRUpdatedPagesPct*.

(See item 1 in Figure 44 on page 853.) *ICRUpdatedPagesPct* is an input parameter of type INTEGER. The default is 1.

ICRChangesPct

Specifies a criterion for recommending an incremental image copy on a table space. If the following condition is true, DSNACCOR recommends an incremental image copy:

The ratio of the number of insert, update, or delete operations since the last image copy, to the total number of rows or LOBs in a table space or partition (expressed as a percentage) is greater than *ICRChangesPct*.

(See item 2 in Figure 44 on page 853.) *ICRChangesPct* is an input parameter of type INTEGER. The default is 1.

CRIndexSize

Specifies, when combined with *CRUpdatedPagesPct* or *CRChangesPct*, a criterion for recommending a full image copy on an index space. (See items 2, 3, and 4 in Figure 43 on page 853.) *CRIndexSize* is an input parameter of type INTEGER. The default is 50.

RRTInsDelUpdPct

Specifies a criterion for recommending that the REORG utility is to be run on a table space. If the following condition is true, DSNACCOR recommends running REORG:

The sum of insert, update, and delete operations since the last REORG, divided by the total number of rows or LOBs in the table space or partition (expressed as a percentage) is greater than *RRTInsDelUpdPct*

(See item 1 in Figure 45 on page 853.) *RRTInsDelUpdPct* is an input parameter of type INTEGER. The default is 20.

RRTUnclustInsPct

Specifies a criterion for recommending that the REORG utility is to be run on a table space. If the following condition is true, DSNACCOR recommends running REORG:

The number of unclustered insert operations, divided by the total number of rows or LOBs in the table space or partition (expressed as a percentage) is greater than *RRTUnclustInsPct*.

(See item 2 in Figure 45 on page 853.) *RRTUnclustInsPct* is an input parameter of type INTEGER. The default is 10.

RRTDisorgLOBPct

Specifies a criterion for recommending that the REORG utility is to be run on a table space. If the following condition is true, DSNACCOR recommends running REORG:

The number of imperfectly chunked LOBs, divided by the total number of rows or LOBs in the table space or partition (expressed as a percentage) is greater than *RRTDisorgLOBPct*.

(See item 3 in Figure 45 on page 853.) *RRTDisorgLOBPct* is an input parameter of type INTEGER. The default is 10.

RRTMassDelLimit

Specifies a criterion for recommending that the REORG utility is to be run on a table space. If one of the following values is greater than *RRTMassDelLimit*, DSNACCOR recommends running REORG:

- The number of mass deletes from a segmented or LOB table space since the last REORG or LOAD REPLACE
- The number of dropped tables from a nonsegmented table space since the last REORG or LOAD REPLACE

(See item 5 in Figure 45 on page 853.) *RRTMassDelLimit* is an input parameter of type INTEGER. The default is 0.

RRTIndRefLimit

Specifies a criterion for recommending that the REORG utility is to be run on a table space. If the following value is greater than *RRTIndRefLimit*, DSNACCOR recommends running REORG:

The total number of overflow records that were created since the last REORG or LOAD REPLACE, divided by the total number of rows or LOBs in the table space or partition (expressed as a percentage)

(See item 4 in Figure 45 on page 853.) *RRTIndRefLimit* is an input parameter of type INTEGER. The default is 10.

RRIInsertDeletePct

Specifies a criterion for recommending that the REORG utility is to be run on an index space. If the following value is greater than *RRIInsertDeletePct*, DSNACCOR recommends running REORG:

The sum of the number of index entries that were inserted and deleted since the last REORG, divided by the total number of index entries in the index space or partition (expressed as a percentage)

(See item 1 in Figure 46 on page 854.) This is an input parameter of type INTEGER. The default is 20.

RRIAppendInsertPct

Specifies a criterion for recommending that the REORG utility is to be run on an index space. If the following value is greater than *RRIAppendInsertPct*, DSNACCOR recommends running REORG:

The number of index entries that were inserted since the last REORG, REBUILD INDEX, or LOAD REPLACE with a key value greater than the maximum key value in the index space or partition, divided by the number of index entries in the index space or partition (expressed as a percentage)

(See item 2 in Figure 46 on page 854.) *RRIInsertDeletePct* is an input parameter of type INTEGER. The default is 10.

RRIPseudoDeletePct

Specifies a criterion for recommending that the REORG utility is to be run on an index space. If the following value is greater than *RRIPseudoDeletePct*, DSNACCOR recommends running REORG:

The number of index entries that were pseudo-deleted since the last REORG, REBUILD INDEX, or LOAD REPLACE, divided by the number of index entries in the index space or partition (expressed as a percentage)

(See item 3 in Figure 46 on page 854.) *RRIPseudoDeletePct* is an input parameter of type INTEGER. The default is 10.

RRIMassDelLimit

Specifies a criterion for recommending that the REORG utility is to be run on an index space. If the number of mass deletes from an index space or partition since the last REORG, REBUILD, or LOAD REPLACE is greater than this value, DSNACCOR recommends running REORG.

(See item 4 in Figure 46 on page 854.) *RRIMassDelLimit* is an input parameter of type INTEGER. The default is 0.

RRILeafLimit

Specifies a criterion for recommending that the REORG utility is to be run on an index space. If the following value is greater than *RRILeafLimit*, DSNACCOR recommends running REORG:

The number of index page splits that occurred since the last REORG, REBUILD INDEX, or LOAD REPLACE that resulted in a large separation between the parts of the original page, divided by the total number of active pages in the index space or partition (expressed as a percentage)

(See item 5 in Figure 46 on page 854.) *RRILeafLimit* is an input parameter of type INTEGER. The default is 10.

RRINumLevelsLimit

Specifies a criterion for recommending that the REORG utility is to be run on an index space. If the following value is greater than *RRINumLevelsLimit*, DSNACCOR recommends running REORG:

The number of levels in the index tree that were added or removed since the last REORG, REBUILD INDEX, or LOAD REPLACE

(See item 6 in Figure 46 on page 854.) *RRINumLevelsLimit* is an input parameter of type INTEGER. The default is 0.

SRTInsDelUpdPct

Specifies, when combined with *SRTInsDelUpdAbs*, a criterion for

recommending that the RUNSTATS utility is to be run on a table space. If both of the following conditions are true, DSNACCOR recommends running RUNSTATS:

- The number of insert, update, or delete operations since the last RUNSTATS on a table space or partition, divided by the total number of rows or LOBs in table space or partition (expressed as a percentage) is greater than *SRTInsDelUpdPct*.
- The sum of the number of inserted and deleted index entries since the last RUNSTATS on an index space or partition is greater than *SRTInsDelUpdAbs*.

(See items 1 and 2 in Figure 47 on page 854.) *SRTInsDelUpdPct* is an input parameter of type INTEGER. The default is 20.

SRTInsDelUpdAbs

Specifies, when combined with *SRTInsDelUpdPct*, a criterion for recommending that the RUNSTATS utility is to be run on a table space. If both of the following conditions are true, DSNACCOR recommends running RUNSTATS:

- The number of insert, update, and delete operations since the last RUNSTATS on a table space or partition, divided by the total number of rows or LOBs in table space or partition (expressed as a percentage) is greater than *SRTInsDelUpdPct*.
- The sum of the number of inserted and deleted index entries since the last RUNSTATS on an index space or partition is greater than *SRTInsDelUpdAbs*.

(See items 1 and 2 in Figure 47 on page 854.) *SRTInsDelUpdAbs* is an input parameter of type INTEGER. The default is 0.

SRTMassDelLimit

Specifies a criterion for recommending that the RUNSTATS utility is to be run on a table space. If the following condition is true, DSNACCOR recommends running RUNSTATS:

- The number of mass deletes from a table space or partition since the last REORG or LOAD REPLACE is greater than *SRTMassDelLimit*.

(See item 3 in Figure 47 on page 854.) *SRTMassDelLimit* is an input parameter of type INTEGER. The default is 0.

SRIInsDelUpdPct

Specifies, when combined with *SRIInsDelUpdAbs*, a criterion for recommending that the RUNSTATS utility is to be run on an index space. If both of the following conditions are true, DSNACCOR recommends running RUNSTATS:

- The number of inserted and deleted index entries since the last RUNSTATS on an index space or partition, divided by the total number of index entries in the index space or partition (expressed as a percentage) is greater than *SRIInsDelUpdPct*.
- The sum of the number of inserted and deleted index entries since the last RUNSTATS on an index space or partition is greater than *SRIInsDelUpdAbs*.

(See items 1 and 2 in Figure 48 on page 854.) *SRIInsDelUpdPct* is an input parameter of type INTEGER. The default is 20.

SRIInsDelUpdAbs

Specifies, when combined with *SRIInsDelUpdPct*, a criterion for recommending that the RUNSTATS utility is to be run on an index space. If the following condition is true, DSNACCOR recommends running RUNSTATS:

- The number of inserted and deleted index entries since the last RUNSTATS on an index space or partition, divided by the total number of index entries in the index space or partition (expressed as a percentage) is greater than *SRInsDelUpdPct*.
- The sum of the number of inserted and deleted index entries since the last RUNSTATS on an index space or partition is greater than *SRInsDelUpdAbs*,

(See items 1 and 2 in Figure 48 on page 854.) *SRInsDelUpdAbs* is an input parameter of type INTEGER. The default is 0.

SRIMassDelLimit

Specifies a criterion for recommending that the RUNSTATS utility is to be run on an index space. If the number of mass deletes from an index space or partition since the last REORG, REBUILD INDEX, or LOAD REPLACE is greater than this value, DSNACCOR recommends running RUNSTATS.

(See item 3 in Figure 48 on page 854.) *SRIMassDelLimit* is an input parameter of type INTEGER. The **default** is 0.

ExtentLimit

Specifies a criterion for recommending that the REORG utility is to be run on a table space or index space. Also specifies that DSNACCOR is to warn the user that the table space or index space has used too many extents. DSNACCOR recommends running REORG, and altering data set allocations if the following condition is true:

- The number of physical extents in the index space, table space, or partition is greater than *ExtentLimit*.

(See Figure 49 on page 854.) *ExtentLimit* is an input parameter of type INTEGER. The default is 50.

LastStatement

When DSNACCOR returns a severe error (return code 12), this field contains the SQL statement that was executing when the error occurred. *LastStatement* is an output parameter of type VARCHAR(8012).

ReturnCode

The return code from DSNACCOR execution. Possible values are:

- | | |
|----|--|
| 0 | DSNACCOR executed successfully. The <i>ErrorMsg</i> parameter contains the approximate percentage of the total number of objects in the subsystem that have information in the real-time statistics tables. |
| 4 | DSNACCOR completed, but one or more input parameters might be incompatible. The <i>ErrorMsg</i> parameter contains the input parameters that might be incompatible. |
| 8 | DSNACCOR terminated with errors. The <i>ErrorMsg</i> parameter contains a message that describes the error. |
| 12 | DSNACCOR terminated with severe errors. The <i>ErrorMsg</i> parameter contains a message that describes the error. The <i>LastStatement</i> parameter contains the SQL statement that was executing when the error occurred. |
| 14 | DSNACCOR terminated because it could not access one or more of the real-time statistics tables. The <i>ErrorMsg</i> parameter contains the names of the tables that DSNACCOR could not access. |
| 15 | DSNACCOR terminated because it encountered a problem with one of the declared temporary tables that it defines and uses. |

16 DSNACCOR terminated because it could not define a declared temporary table. No table spaces were defined in the TEMP database.

NULL DSNACCOR terminated but could not set a return code.

ReturnCode is an output parameter of type INTEGER.

ErrorMsg

Contains information about DSNACCOR execution. If DSNACCOR runs successfully (*ReturnCode*=0), this field contains the approximate percentage of objects in the subsystem that are in the real-time statistics tables. Otherwise, this field contains error messages. *ErrorMsg* is an output parameter of type VARCHAR(1331).

IFCARetCode

Contains the return code from an IFI COMMAND call. DSNACCOR issues commands through the IFI interface to determine the status of objects. *IFCARetCode* is an output parameter of type INTEGER.

IFCAResCode

Contains the reason code from an IFI COMMAND call. *IFCAResCode* is an output parameter of type INTEGER.

ExcessBytes

Contains the number of bytes of information that did not fit in the IFI return area after an IFI COMMAND call. *ExcessBytes* is an output parameter of type INTEGER.

DSNACCOR formulas for recommending actions

The following formulas specify the criteria that DSNACCOR uses for its recommendations and warnings. The variables in italics are DSNACCOR input parameters. The capitalized variables are columns of the SYSIBM.SYSTABLESPACESTATS or SYSIBM.SYSINDEXSPACESTATS tables. The numbers to the right of selected items are reference numbers for the option descriptions in “Option descriptions” on page 844.

The figure below shows the formula that DSNACCOR uses to recommend a full image copy on a table space.

```
((QueryType='COPY' OR QueryType='ALL') AND
(ObjectType='TS' OR ObjectType='ALL') AND
ICType='F') AND
(COPYLASTTIME IS NULL OR
REORGLASTTIME>COPYLASTTIME OR
LOADRLASTTIME>COPYLASTTIME OR
(CURRENT DATE-COPYLASTTIME)>CRDaySncLastCopy OR 1
(COPYUPDATEDPAGES*100)/NACTIVE>CRUpdatedPagesPct OR 2
(COPYCHANGES*100)/TOTALROWS>CRChangesPct) 3
```

Figure 42. DSNACCOR formula for recommending a full image copy on a table space

The figure below shows the formula that DSNACCOR uses to recommend a full image copy on an index space.

```

((QueryType='COPY' OR QueryType='ALL') AND
(ObjectType='IX' OR ObjectType='ALL') AND
(ICType='F' OR ICType='B')) AND
(COPYLASTTIME IS NULL OR
REORGLASTTIME>COPYLASTTIME OR
LOADRLASTTIME>COPYLASTTIME OR
REBUILDLASTTIME>COPYLASTTIME OR
(CURRENT DATE-COPYLASTTIME)>CRDaySncLastCopy OR
(NACTIVE>CRIndexSize AND
((COPYUPDATEDPAGES*100)/NACTIVE>CRUpdatedPagesPct OR
(COPYCHANGES*100)/TOTALENTRIES>CRChangesPct))

```

1
2
3
4

Figure 43. DSNACCOR formula for recommending a full image copy on an index space

The figure below shows the formula that DSNACCOR uses to recommend an incremental image copy on a table space.

```

((QueryType='COPY' OR QueryType='ALL') AND
(ObjectType='TS' OR ObjectType='ALL') AND
ICType='I' AND
COPYLASTTIME IS NOT NULL) AND
(LOADRLASTTIME>COPYLASTTIME OR
REORGLASTTIME>COPYLASTTIME OR
(COPYUPDATEDPAGES*100)/NACTIVE>ICRUpdatedPagesPct OR
(COPYCHANGES*100)/TOTALROWS>ICRChangesPct))

```

1
2

Figure 44. DSNACCOR formula for recommending an incremental image copy on a table space

The figure below shows the formula that DSNACCOR uses to recommend a REORG on a table space. If the table space is a LOB table space, and CHCKLVL=1, the formula does not include EXTENTS>ExtentLimit.

```

((QueryType='REORG' OR QueryType='ALL') AND
(ObjectType='TS' OR ObjectType='ALL')) AND
(REORGLASTTIME IS NULL OR
((REORGINSERTS+REORGDELETES+REORGUPDATES)*100)/TOTALROWS>RRTInsDelUpdPct OR
(REORGUNCLUSTINS*100)/TOTALROWS>RRTUnclustInsPct OR
(REORGDISORGL*100)/TOTALROWS>RRTDisorgLOBPct OR
((REORGNearIndRef+REORGFarIndRef)*100)/TOTALROWS>RRTIndRefLimit OR
REORGMASDELETE>RRTMassDelLimit OR
EXTENTS>ExtentLimit)

```

1
2
3
4
5
6

Figure 45. DSNACCOR formula for recommending a REORG on a table space

The figure below shows the formula that DSNACCOR uses to recommend a REORG on an index space.

```

((QueryType='REORG' OR QueryType='ALL') AND
(ObjectType='IX' OR ObjectType='ALL')) AND
(REORGLASTTIME IS NULL OR
((REORGINSERTS+REORGDELETES)*100)/TOTALENTRIES>RRIInsertDeletePct OR
(REORGAPPENDINSERT*100)/TOTALENTRIES>RRIAppendInsertPct OR
(REORGPSEUDODELETES*100)/TOTALENTRIES>RRIPseudoDeletePct OR
REORGMASDELETE>RRIMassDeleteLimit OR
(REORGLEAFFAR*100)/NACTIVE>RRILeafLimit OR
REORGNUMLEVELS>RRINumLevelsLimit OR
EXTENTS>ExtentLimit)

```

1
2
3
4
5
6
7

Figure 46. DSNACCOR formula for recommending a REORG on an index space

The figure below shows the formula that DSNACCOR uses to recommend RUNSTATS on a table space.

```

((QueryType='RUNSTATS' OR QueryType='ALL') AND
(ObjectType='TS' OR ObjectType='ALL')) AND
(STATSLASTTIME IS NULL OR
(((STATSINSERTS+STATSDELETES+STATSUPDATES)*100)/TOTALROWS>SRTInsDelUpdPct AND
(STATSINSERTS+STATSDELETES+STATSUPDATES)>SRTInsDelUpdAbs) OR
STATSMASDELETE>SRTMassDeleteLimit)

```

1
2
3

Figure 47. DSNACCOR formula for recommending RUNSTATS on a table space

The figure below shows the formula that DSNACCOR uses to recommend RUNSTATS on an index space.

```

((QueryType='RUNSTATS' OR QueryType='ALL') AND
(ObjectType='IX' OR ObjectType='ALL')) AND
(STATSLASTTIME IS NULL OR
(((STATSINSERTS+STATSDELETES)*100)/TOTALENTRIES>SRIInsDelUpdPct AND
(STATSINSERTS+STATSDELETES)>SRIInsDelUpdPct) OR
STATSMASDELETE>SRIInsDelUpdAbs)

```

1
2
3

Figure 48. DSNACCOR formula for recommending RUNSTATS on an index space

The figure below shows the formula that DSNACCOR uses to that too many index space or table space extents have been used.

```

EXTENTS>ExtentLimit

```

Figure 49. DSNACCOR formula for warning that too many data set extents for a table space or index space are used

Using an exception table

An exception table is an optional, user-created DB2 table that you can use to place information in the INEXCEPTTABLE column of the recommendations result set. You can put any information in the INEXCEPTTABLE column, but the most common use of this column is to filter the recommendations result set. Each row in the exception table represents an object for which you want to provide information for the recommendations result set.

To create the exception table, execute a CREATE TABLE statement similar to the following one. You can include other columns in the exception table, but you must include at least the columns that are shown.

```
CREATE TABLE DSNACC.EXCEPT_TBL
  (DBNAME CHAR(8) NOT NULL,
   NAME CHAR(8) NOT NULL,
   QUERYTYPE CHAR(40))
CCSID EBCDIC;
```

The meanings of the columns are:

DBNAME

The database name for an object in the exception table.

NAME

The table space name or index space name for an object in the exception table.

QUERYTYPE

The information that you want to place in the INEXCEPTTABLE column of the recommendations result set.

If you put a null value in this column, DSNACCOR puts the value YES in the INEXCEPTTABLE column of the recommendations result set row for the object that matches the DBNAME and NAME values.

Recommendation: If you plan to put many rows in the exception table, create a nonunique index on DBNAME, NAME, and QUERYTYPE.

After you create the exception table, insert a row for each object for which you want to include information in the INEXCEPTTABLE column.

Example: Suppose that you want the INEXCEPTTABLE column to contain the string 'IRRELEVANT' for table space STAFF in database DSNDB04. You also want the INEXCEPTTABLE column to contain 'CURRENT' for table space DSN8S10D in database DSN8D10A. Execute these INSERT statements:

```
INSERT INTO DSNACC.EXCEPT_TBL VALUES('DSNDB04 ', 'STAFF ', 'IRRELEVANT');
INSERT INTO DSNACC.EXCEPT_TBL VALUES('DSN8D10A', 'DSN8S10D', 'CURRENT');
```

To use the contents of INEXCEPTTABLE for filtering, include a condition that involves the INEXCEPTTABLE column in the search condition that you specify in your *Criteria* input parameter.

Example: Suppose that you want to include all rows for database DSNDB04 in the recommendations result set, except for those rows that contain the string 'IRRELEVANT' in the INEXCEPTTABLE column. You might include the following search condition in your *Criteria* input parameter:

```
DBNAME='DSNDB04' AND INEXCEPTTABLE<>'IRRELEVANT'
```

Example

The following COBOL example that shows variable declarations and an SQL CALL for obtaining recommendations for objects in databases DSN8D10A and DSN8D10L. This example also outlines the steps that you need to perform to retrieve the two result sets that DSNACCOR returns.

```
WORKING-STORAGE SECTION.
:
*****
* DSNACCOR PARAMETERS *
*****
```

```

01 QUERYTYPE.
   49 QUERYTYPE-LN          PICTURE S9(4) COMP VALUE 40.
   49 QUERYTYPE-DTA        PICTURE X(40)  VALUE 'ALL'.
01 OBJECTTYPE.
   49 OBJECTTYPE-LN         PICTURE S9(4) COMP VALUE 3.
   49 OBJECTTYPE-DTA        PICTURE X(3)   VALUE 'ALL'.
01 ICTYPE.
   49 ICTYPE-LN             PICTURE S9(4) COMP VALUE 1.
   49 ICTYPE-DTA            PICTURE X(1)   VALUE 'B'.
01 STATSSHEMA.
   49 STATSSHEMA-LN         PICTURE S9(4) COMP VALUE 128.
   49 STATSSHEMA-DTA        PICTURE X(128)  VALUE 'SYSIBM'.
01 CATLGSHEMA.
   49 CATLGSHEMA-LN         PICTURE S9(4) COMP VALUE 128.
   49 CATLGSHEMA-DTA        PICTURE X(128)  VALUE 'SYSIBM'.
01 LOCALSCHEMA.
   49 LOCALSCHEMA-LN        PICTURE S9(4) COMP VALUE 128.
   49 LOCALSCHEMA-DTA       PICTURE X(128)  VALUE 'DSNACC'.
01 CHKLVL
   49 CHKLVL                 PICTURE S9(9) COMP VALUE +3.
01 CRITERIA.
   49 CRITERIA-LN           PICTURE S9(4) COMP VALUE 4096.
   49 CRITERIA-DTA         PICTURE X(4096)  VALUE SPACES.
01 RESTRICTED.
   49 RESTRICTED-LN         PICTURE S9(4) COMP VALUE 80.
   49 RESTRICTED-DTA        PICTURE X(80)   VALUE SPACES.
01 CRUPDATEDPAGESPCT
01 CRCHANGESPCT
01 CRDAYSNCLASTCOPY
01 ICRUPDATEDPAGESPCT
01 ICRCHANGESPCT
01 CRINDEXSIZE
01 RRTINSDELUPDPCT
01 RRTUNCLUSTINSPCT
01 RRTDISORGLBOPCT
01 RRTMASSDELLIMIT
01 RRTINDREFLIMIT
01 RRIINSERTDELETEPCT
01 RRIAPPENDINSERTPCT
01 RRIPEUDODELETEPCT
01 RRIMASSDELLIMIT
01 RRILEAFLIMIT
01 RRINUMLEVELSLIMIT
01 SRTINSDELUPDPCT
01 SRTINSDELUPDABS
01 SRTMASSDELLIMIT
01 SRIINSDELUPDPCT
01 SRIINSDELUPDABS
01 SRIMASSDELLIMIT
01 EXTENTLIMIT
01 LASTSTATEMENT.
   49 LASTSTATEMENT-LN      PICTURE S9(4) COMP VALUE 8012.
   49 LASTSTATEMENT-DTA     PICTURE X(8012)  VALUE SPACES.
01 RETURNCODE
   49 RETURNCODE             PICTURE S9(9) COMP VALUE +0.
01 ERRORMSG.
   49 ERRORMSG-LN           PICTURE S9(4) COMP VALUE 1331.
   49 ERRORMSG-DTA          PICTURE X(1331)  VALUE SPACES.
01 IFCARETCODE
01 IFCARESCODE
01 EXCESSBYTES
   49 EXCESSBYTES            PICTURE S9(9) COMP VALUE +0.

*****
* INDICATOR VARIABLES.          *
* INITIALIZE ALL NON-ESSENTIAL INPUT *
* VARIABLES TO -1, TO INDICATE THAT THE *
* INPUT VALUE IS NULL.          *
*****
01 QUERYTYPE-IND          PICTURE S9(4) COMP-4 VALUE +0.
01 OBJECTTYPE-IND         PICTURE S9(4) COMP-4 VALUE +0.

```

```

01 ICTYPE-IND          PICTURE S9(4) COMP-4 VALUE +0.
01 STATSSHEMA-IND     PICTURE S9(4) COMP-4 VALUE -1.
01 CATLGSCHEMA-IND   PICTURE S9(4) COMP-4 VALUE -1.
01 LOCALSCHEMA-IND   PICTURE S9(4) COMP-4 VALUE -1.
01 CHKLVL-IND        PICTURE S9(4) COMP-4 VALUE -1.
01 CRITERIA-IND      PICTURE S9(4) COMP-4 VALUE -1.
01 RESTRICTED-IND    PICTURE S9(4) COMP-4 VALUE -1.
01 CRUPDATEDPAGEPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 CRCHANGESPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 CRDAYSNCCLASTCOPY-IND PICTURE S9(4) COMP-4 VALUE -1.
01 ICRUPDATEDPAGEPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 ICRCHANGESPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 CRINDEXSIZE-IND   PICTURE S9(4) COMP-4 VALUE -1.
01 RRTINSDELUPDPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRTUNCLUSTINS PCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRTDISORGL OBPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRTMASSDELLIMIT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRTINDREFLIMIT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRIINSERTDELETEPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRIAPPENDINSERTPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRIPEUDODELETEPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRIASSDELLIMIT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 RRILEAFLIMIT-IND  PICTURE S9(4) COMP-4 VALUE -1.
01 RRINUMLEVELSLIMIT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 SRTINSDELUPDPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 SRTINSDELUPDABS-IND PICTURE S9(4) COMP-4 VALUE -1.
01 SRTMASSDELLIMIT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 SRIINSDELUPDPCT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 SRIINSDELUPDABS-IND PICTURE S9(4) COMP-4 VALUE -1.
01 SRIMASSDELLIMIT-IND PICTURE S9(4) COMP-4 VALUE -1.
01 EXTENTLIMIT-IND   PICTURE S9(4) COMP-4 VALUE -1.
01 LASTSTATEMENT-IND PICTURE S9(4) COMP-4 VALUE +0.
01 RETURNCODE-IND    PICTURE S9(4) COMP-4 VALUE +0.
01 ERRORMSG-IND      PICTURE S9(4) COMP-4 VALUE +0.
01 IFCARETCODE-IND   PICTURE S9(4) COMP-4 VALUE +0.
01 IFCARESCODE-IND   PICTURE S9(4) COMP-4 VALUE +0.
01 EXCESSBYTES-IND   PICTURE S9(4) COMP-4 VALUE +0.

```

PROCEDURE DIVISION.

:

```

*****
* SET VALUES FOR DSNACCOR INPUT PARAMETERS: *
* - USE THE CHKLVL PARAMETER TO CAUSE DSNACCOR TO CHECK *
* FOR ORPHANED OBJECTS AND INDEX SPACES WITHOUT *
* TABLE SPACES, BUT INCLUDE THOSE OBJECTS IN THE *
* RECOMMENDATIONS RESULT SET (CHKLVL=1+2+16=19) *
* - USE THE CRITERIA PARAMETER TO CAUSE DSNACCOR TO *
* MAKE RECOMMENDATIONS ONLY FOR OBJECTS IN DATABASES *
* DSN8D10A AND DSN8D10L. *
* - FOR THE FOLLOWING PARAMETERS, SET THESE VALUES, *
* WHICH ARE LOWER THAN THE DEFAULTS: *
* CRUPDATEDPAGEPCT 4 *
* CRCHANGESPCT 2 *
* RRTINSDELUPDPCT 2 *
* RRTUNCLUSTINS PCT 5 *
* RRTDISORGL OBPCT 5 *
* RRIAPPENDINSERTPCT 5 *
* SRTINSDELUPDPCT 5 *
* SRIINSDELUPDPCT 5 *
* EXTENTLIMIT 3 *
*****

```

```

MOVE 19 TO CHKLVL.
MOVE SPACES TO CRITERIA-DTA.
MOVE 'DBNAME = 'DSN8D10A' OR DBNAME = 'DSN8D10L''
TO CRITERIA-DTA.
MOVE 46 TO CRITERIA-LN.
MOVE 4 TO CRUPDATEDPAGEPCT.

```

```

MOVE 2 TO CRCHANGESPCT.
MOVE 2 TO RRTINSDELUPDPCT.
MOVE 5 TO RRTUNCLUSTINSPECT.
MOVE 5 TO RRTDISORGLBPCT.
MOVE 5 TO RRIAPPENDINSERTPCT.
MOVE 5 TO SRTINSDELUPDPCT.
MOVE 5 TO SRIINSDELUPDPCT.
MOVE 3 TO EXTENTLIMIT.
*****
* INITIALIZE OUTPUT PARAMETERS *
*****
MOVE SPACES TO LASTSTATEMENT-DTA.
MOVE 1 TO LASTSTATEMENT-LN.
MOVE 0 TO RETURNCODE-02.
MOVE SPACES TO ERRORMSG-DTA.
MOVE 1 TO ERRORMSG-LN.
MOVE 0 TO IFCARETCODE.
MOVE 0 TO IFCARESCODE.
MOVE 0 TO EXCESSBYTES.
*****
* SET THE INDICATOR VARIABLES TO 0 FOR NON-NULL INPUT *
* PARAMETERS (PARAMETERS FOR WHICH YOU DO NOT WANT *
* DSNACCOR TO USE DEFAULT VALUES) AND FOR OUTPUT *
* PARAMETERS. *
*****
MOVE 0 TO CHKLVL-IND.
MOVE 0 TO CRITERIA-IND.
MOVE 0 TO CRUPDATEDPAGESPCT-IND.
MOVE 0 TO CRCHANGESPCT-IND.
MOVE 0 TO RRTINSDELUPDPCT-IND.
MOVE 0 TO RRTUNCLUSTINSPECT-IND.
MOVE 0 TO RRTDISORGLBPCT-IND.
MOVE 0 TO RRIAPPENDINSERTPCT-IND.
MOVE 0 TO SRTINSDELUPDPCT-IND.
MOVE 0 TO SRIINSDELUPDPCT-IND.
MOVE 0 TO EXTENTLIMIT-IND.
MOVE 0 TO LASTSTATEMENT-IND.
MOVE 0 TO RETURNCODE-IND.
MOVE 0 TO ERRORMSG-IND.
MOVE 0 TO IFCARETCODE-IND.
MOVE 0 TO IFCARESCODE-IND.
MOVE 0 TO EXCESSBYTES-IND.
:
:
*****
* CALL DSNACCOR *
*****
EXEC SQL
CALL SYSPROC.DSNACCOR
(:QUERYTYPE           :QUERYTYPE-IND,
:OBJECTTYPE           :OBJECTTYPE-IND,
:ICTYPE               :ICTYPE-IND,
:STATSSHEMA           :STATSSHEMA-IND,
:CATLGSHEMA           :CATLGSHEMA-IND,
:LOCALSCHEMA          :LOCALSCHEMA-IND,
:CHKLVL               :CHKLVL-IND,
:CRITERIA              :CRITERIA-IND,
:RESTRICTED           :RESTRICTED-IND,
:CRUPDATEDPAGESPCT   :CRUPDATEDPAGESPCT-IND,
:CRCHANGESPCT         :CRCHANGESPCT-IND,
:CRDAYSNCLASTCOPY     :CRDAYSNCLASTCOPY-IND,
:ICRUPDATEDPAGESPCT   :ICRUPDATEDPAGESPCT-IND,
:ICRCHANGESPCT        :ICRCHANGESPCT-IND,
:CRINDEXSIZE          :CRINDEXSIZE-IND,
:RRTINSDELUPDPCT      :RRTINSDELUPDPCT-IND,
:RRTUNCLUSTINSPECT    :RRTUNCLUSTINSPECT-IND,
:RRTDISORGLBPCT       :RRTDISORGLBPCT-IND,
:RTMASSEDELLIMIT     :RTMASSEDELLIMIT-IND,

```

```

:RRTINDREFLIMIT      :RRTINDREFLIMIT-IND,
:RRIINSERTDELETEPCT :RRIINSERTDELETEPCT-IND,
:RRIAPPENDINSERTPCT :RRIAPPENDINSERTPCT-IND,
:RRIPSEUDODELETEPCT :RRIPSEUDODELETEPCT-IND,
:RRIMASSDELLIMIT    :RRIMASSDELLIMIT-IND,
:RRILEAFLIMIT       :RRILEAFLIMIT-IND,
:RRINUMLEVELSLIMIT :RRINUMLEVELSLIMIT-IND,
:SRTINSDELUPDPCT    :SRTINSDELUPDPCT-IND,
:SRTINSDELUPDABS    :SRTINSDELUPDABS-IND,
:SRTMASSDELLIMIT    :SRTMASSDELLIMIT-IND,
:SRIINSDELUPDPCT    :SRIINSDELUPDPCT-IND,
:SRIINSDELUPDABS    :SRIINSDELUPDABS-IND,
:SRIMASSDELLIMIT    :SRIMASSDELLIMIT-IND,
:EXTENTLIMIT        :EXTENTLIMIT-IND,
:LASTSTATEMENT      :LASTSTATEMENT-IND,
:RETURNCODE         :RETURNCODE-IND,
:ERRORMSG           :ERRORMSG-IND,
:IFCARETCODE        :IFCARETCODE-IND,
:IFCARESCODE        :IFCARESCODE-IND,
:EXCESSBYTES        :EXCESSBYTES-IND)
END-EXEC.
*****
* ASSUME THAT THE SQL CALL RETURNED +466, WHICH MEANS THAT *
* RESULT SETS WERE RETURNED. RETRIEVE RESULT SETS.          *
*****
* LINK EACH RESULT SET TO A LOCATOR VARIABLE
  EXEC SQL ASSOCIATE LOCATORS (:LOC1, :LOC2)
  WITH PROCEDURE SYSPROC.DSNACCOR
  END-EXEC.
* LINK A CURSOR TO EACH RESULT SET
  EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :LOC1
  END-EXEC.
  EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :LOC2
  END-EXEC.
* PERFORM FETCHES USING C1 TO RETRIEVE ALL ROWS FROM FIRST RESULT SET
* PERFORM FETCHES USING C2 TO RETRIEVE ALL ROWS FROM SECOND RESULT SET

```

Figure 50. Example of DSNACCOR invocation

Output

If DSNACCOR executes successfully, in addition to the output parameters described in “Option descriptions” on page 844, DSNACCOR returns two result sets.

The first result set contains the results from IFI COMMAND calls that DSNACCOR makes. The following table shows the format of the first result set.

Table 136. Result set row for first DSNACCOR result set

Column name	Data type	Contents
RS_SEQUENCE	INTEGER	Sequence number of the output line
RS_DATA	CHAR(80)	A line of command output

The result set contains rows for table spaces, index spaces, or partitions, if both of the following conditions are true for the object:

- If the *Criteria* input parameter contains a search condition, and the search condition is true for the table space, index space, or partition.
- DSNACCOR recommends at least one action for the table space, index space, or partition.

The result set contains one row for each nonpartitioned table space or nonpartitioning index space. For partitioned table spaces or partitioning indexes, the result set contains one row for each partition. If *ChkLvl* 8 is specified, the result set might contain additional rows, including duplicate rows for the same object.

The following table shows the columns of a result set row.

Table 137. Result set row for second DSNACCOR result set

Column name	Data type	Description
DBNAME	CHAR(8)	Name of the database that contains the object.
NAME	CHAR(8)	Table space or index space name.
PARTITION	INTEGER	Data set number or partition number.
OBJECTTYPE	CHAR(2)	DB2 object type: <ul style="list-style-type: none"> • TS for a table space • IX for an index space
OBJECTSTATUS	CHAR(36)	Status of the object: <ul style="list-style-type: none"> • ORPHANED, if the object is an index space with no corresponding table space, or if the object does not exist • If the object is in a restricted state, one of the following values: <ul style="list-style-type: none"> – TS=<i>restricted-state</i>, if OBJECTTYPE is TS – IX=<i>restricted-state</i>, if OBJECTTYPE is IX <i>restricted-state</i> is one of the status codes that appear in DISPLAY DATABASE output. <p>Related information:</p> <ul style="list-style-type: none"> DSNT362I (DB2 Messages) -DISPLAY DATABASE (DB2) (DB2 Commands) • A, if the object is in an advisory state. • L, if the object is a logical partition, but not in an advisory state. • AL, if the object is a logical partition and in an advisory state.
IMAGECOPY	CHAR(3)	COPY recommendation: <ul style="list-style-type: none"> • If OBJECTTYPE is TS: FUL (full image copy), INC (incremental image copy), or NO • If OBJECTTYPE is IX: YES or NO
RUNSTATS	CHAR(3)	RUNSTATS recommendation: YES or NO.
EXTENTS	CHAR(3)	Indicates whether the data sets for the object have exceeded <i>ExtentLimit</i> : YES or NO.
REORG	CHAR(3)	REORG recommendation: YES or NO.
INEXCEPTABLE	CHAR(40)	A string that contains one of the following values: <ul style="list-style-type: none"> • Text that you specify in the QUERYTYPE column of the exception table. • YES, if you put a row in the exception table for the object that this result set row represents, but you specify NULL in the QUERYTYPE column. • NO, if the exception table exists but does not have a row for the object that this result set row represents. • Null, if the exception table does not exist, or if the <i>ChkLvl</i> input parameter does not include the value 4.

Table 137. Result set row for second DSNACCOR result set (continued)

Column name	Data type	Description
ASSOCIATEDTS	CHAR(8)	If OBJECTTYPE is IX and the <i>ChkLvl</i> input parameter includes the value 2, this value is the name of the table space that is associated with the index space. Otherwise null.
COPYLASTTIME	TIMESTAMP	Timestamp of the last full or incremental image copy on the object. Null if COPY was never run, or if the last COPY execution was terminated.
LOADRLASTTIME	TIMESTAMP	Timestamp of the last LOAD REPLACE on the object. Null if LOAD REPLACE was never run, or if the last LOAD REPLACE execution was terminated.
REBUILDLASTTIME	TIMESTAMP	Timestamp of the last REBUILD INDEX on the object. Null if REBUILD INDEX was never run, or if the last REBUILD INDEX execution was terminated.
CRUPDPGSPCT	INTEGER	If OBJECTTYPE is TS or IX and IMAGECOPY is YES, the ratio of distinct updated pages to preformatted pages, expressed as a percentage. Otherwise null.
CRCPYCHGPCT	INTEGER	If OBJECTTYPE is TS and IMAGECOPY is YES, the ratio of the total number insert, update, and delete operations since the last image copy to the total number of rows or LOBs in the table space or partition, expressed as a percentage. If OBJECTTYPE is IX and IMAGECOPY is YES, the ratio of the total number of insert and delete operations since the last image copy to the total number of entries in the index space or partition, expressed as a percentage. Otherwise null.
CRDAYSCELSTCPY	INTEGER	If OBJECTTYPE is TS or IX and IMAGECOPY is YES, the number of days since the last image copy. Otherwise null.
CRINDEXSIZE	INTEGER	If OBJECTTYPE is IX and IMAGECOPY is YES, the number of active pages in the index space or partition. Otherwise null.
REORGLASTTIME	TIMESTAMP	Timestamp of the last REORG on the object. Null if REORG was never run, or if the last REORG execution was terminated.
RRTINSDELUPDPCT	INTEGER	If OBJECTTYPE is TS and REORG is YES, the ratio of the sum of insert, update, and delete operations since the last REORG to the total number of rows or LOBs in the table space or partition, expressed as a percentage. Otherwise null.
RRTUNCINS PCT	INTEGER	If OBJECTTYPE is TS and REORG is YES, the ratio of the number of unclustered insert operations to the total number of rows or LOBs in the table space or partition, expressed as a percentage. Otherwise null.
RRTDISORGLBPCT	INTEGER	If OBJECTTYPE is TS and REORG is YES, the ratio of the number of imperfectly chunked LOBs to the total number of rows or LOBs in the table space or partition, expressed as a percentage. Otherwise null.
RRTMASSDELETE	INTEGER	If OBJECTTYPE is TS, REORG is YES, and the table space is a segmented table space or LOB table space, the number of mass deletes since the last REORG or LOAD REPLACE. If OBJECTTYPE is TS, REORG is YES, and the table space is nonsegmented, the number of dropped tables since the last REORG or LOAD REPLACE. Otherwise null.
RRTINDREF	INTEGER	If OBJECTTYPE is TS, REORG is YES, the ratio of the total number of overflow records that were created since the last REORG or LOAD REPLACE to the total number of rows or LOBs in the table space or partition, expressed as a percentage. Otherwise null.

Table 137. Result set row for second DSNACCOR result set (continued)

Column name	Data type	Description
RRIINSDELPCT	INTEGER	If OBJECTTYPE is IX and REORG is YES, the ratio of the total number of insert and delete operations since the last REORG to the total number of index entries in the index space or partition, expressed as a percentage. Otherwise null.
RRIAPPINSPECT	INTEGER	If OBJECTTYPE is IX and REORG is YES, the ratio of the number of index entries that were inserted since the last REORG, REBUILD INDEX, or LOAD REPLACE that had a key value greater than the maximum key value in the index space or partition, to the number of index entries in the index space or partition, expressed as a percentage. Otherwise null.
RRIPSDELDPCT	INTEGER	If OBJECTTYPE is IX and REORG is YES, the ratio of the number of index entries that were pseudo-deleted (the RID entry was marked as deleted) since the last REORG, REBUILD INDEX, or LOAD REPLACE to the number of index entries in the index space or partition, expressed as a percentage. Otherwise null.
RRIMASSDELETE	INTEGER	If OBJECTTYPE is IX and REORG is YES, the number of mass deletes from the index space or partition since the last REORG, REBUILD, or LOAD REPLACE. Otherwise null.
RRILEAF	INTEGER	If OBJECTTYPE is IX and REORG is YES, the ratio of the number of index page splits that occurred since the last REORG, REBUILD INDEX, or LOAD REPLACE in which the higher part of the split page was far from the location of the original page, to the total number of active pages in the index space or partition, expressed as a percentage. Otherwise null.
RRINUMLEVELS	INTEGER	If OBJECTTYPE is IX and REORG is YES, the number of levels in the index tree that were added or removed since the last REORG, REBUILD INDEX, or LOAD REPLACE. Otherwise null.
STATSLASTTIME	TIMESTAMP	Timestamp of the last RUNSTATS on the object. Null if RUNSTATS was never run, or if the last RUNSTATS execution was terminated.
SRTINSDELUPDPCT	INTEGER	If OBJECTTYPE is TS and RUNSTATS is YES, the ratio of the total number of insert, update, and delete operations since the last RUNSTATS on a table space or partition, to the total number of rows or LOBs in the table space or partition, expressed as a percentage. Otherwise null.
SRTINSDELUPDABS	INTEGER	If OBJECTTYPE is TS and RUNSTATS is YES, the total number of insert, update, and delete operations since the last RUNSTATS on a table space or partition. Otherwise null.
SRTMASSDELETE	INTEGER	If OBJECTTYPE is TS and RUNSTATS is YES, the number of mass deletes from the table space or partition since the last REORG or LOAD REPLACE. Otherwise null.
SRIINSDELPCT	INTEGER	If OBJECTTYPE is IX and RUNSTATS is YES, the ratio of the total number of insert and delete operations since the last RUNSTATS on the index space or partition, to the total number of index entries in the index space or partition, expressed as a percentage. Otherwise null.
SRIINSDELABS	INTEGER	If OBJECTTYPE is IX and RUNSTATS is YES, the number insert and delete operations since the last RUNSTATS on the index space or partition. Otherwise null.

Table 137. Result set row for second DSNACCOR result set (continued)

Column name	Data type	Description
SRIMASSDELETE	INTEGER	If OBJECTTYPE is IX and RUNSTATS is YES, the number of mass deletes from the index space or partition since the last REORG, REBUILD INDEX, or LOAD REPLACE. Otherwise, this value is null.
TOTALEXTENTS	SMALLINT	If EXTENTS is YES, the number of physical extents in the table space, index space, or partition. Otherwise, this value is null.



Related reference:

CREATE DATABASE (DB2 SQL)

CREATE TABLESPACE (DB2 SQL)

XSR_REGISTER stored procedure

The XSR_REGISTER procedure is the first stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XSR.

The user that calls this stored procedure is considered the creator of this XML schema. DB2 obtains the namespace attribute from the schema document when XSR_COMPLETE is invoked.

Environment for XSR_REGISTER

XSR_REGISTER runs in a WLM-established stored procedures address space.

|
|
|
|

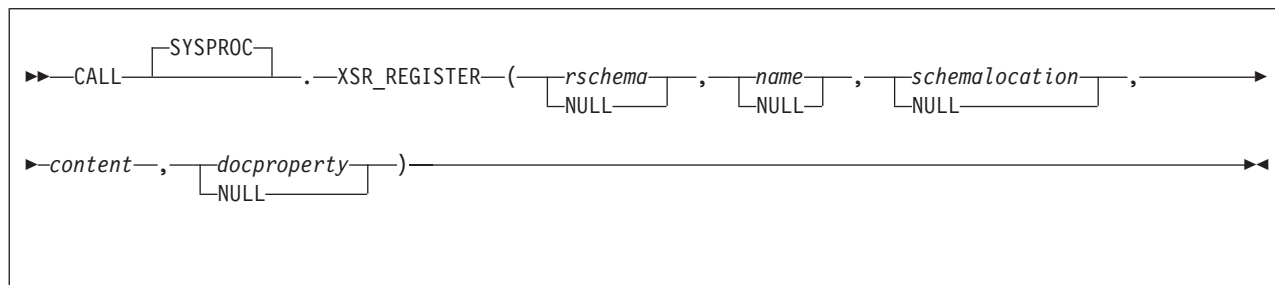
Recommendation: Use WLM environment DSNWLM_XML for running XSR_REGISTER. The startup procedure for this environment, DSNWLMX, can be configured during installation, using installation panel DSNTIPRJ, XML schema processing routines.

Authorization required for XSR_REGISTER

The user ID of the caller of the procedure must have the EXECUTE privilege on the XSR_REGISTER stored procedure.

XSR_REGISTER syntax diagram

The following syntax diagram shows the CALL statement for invoking XSR_REGISTER.



XSR_REGISTER option descriptions

rschema

An input argument of type VARCHAR(128) that specifies the SQL schema for the XML schema. If a value is specified, it must be SYSXSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name

An input and output argument of type VARCHAR(128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is "rschema.name" and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation

An input argument of type VARCHAR(1000), which can have a NULL value, that indicates the schema location of the primary XML schema document. This argument is the "external name" of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content

An input parameter of type BLOB(30M) that contains the content of the primary XML schema document. This argument cannot have a NULL value; an XML schema document must be supplied. The content of the XML schema document must be encoded in Unicode.

docproperty

An input parameter of type BLOB(5M) that indicates the properties for the primary XML schema document. This parameter can have a NULL value; otherwise, the value is an XML document.

Example of XSR_REGISTER

The following example calls the XSR_REGISTER stored procedure:

```
CALL SYSPROC.XSR_REGISTER(  
  'SYSXSR',  
  'POschema',  
  'http://myPOschema/PO.xsd',  
  :content_host_var,  
  :docproperty_host_var)
```

In this example, XSR_REGISTER folds the name POschema to uppercase, so the registered schema name is POSHEMA. If you do not want XSR_REGISTER to fold POschema to uppercase, you need to delimit the name with double quotation marks ("), as in the following example.

```
CALL SYSPROC.XSR_REGISTER(  
  'SYSXSR',  
  '"POschema"',  
  'http://myPOschema/PO.xsd',  
  :content_host_var,  
  :docproperty_host_var)
```

Related concepts:

 Command line processor (DB2 Commands)

Example of XML schema registration and removal using stored procedures (DB2 Programming for XML)

XSR_ADDSCHEMADOC stored procedure

The XSR_ADDSCHEMADOC stored procedure adds every XML schema, other than the primary XML schema document, to the XSR.

Each XML schema in the XSR can consist of one or more XML schema documents. When an XML schema consists of multiple documents, you need to call XSR_ADDSCHEMADOC for the additional documents.

Environment for XSR_ADDSCHEMADOC

XSR_ADDSCHEMADOC runs in a WLM-established stored procedures address space.

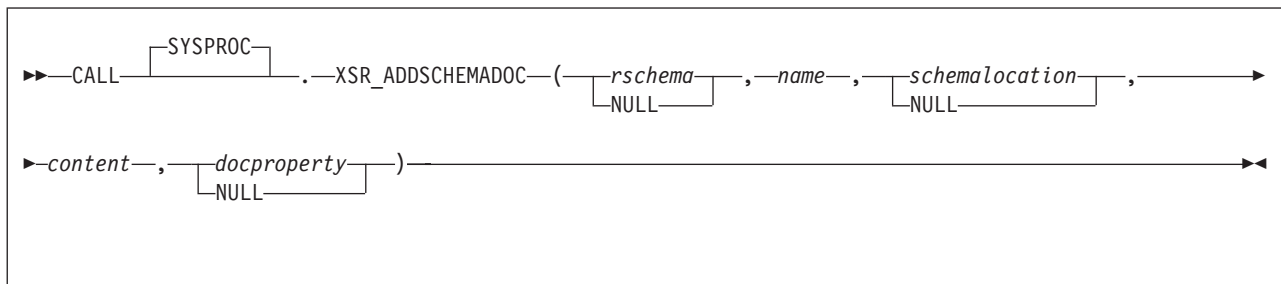
Recommendation: Use WLM environment DSNWLM_XML for running XSR_ADDSCHEMADOC. The startup procedure for this environment, DSNWLMX, can be configured during installation, using installation panel DSNTIPRJ, XML schema processing routines.

Authorization required for XSR_ADDSCHEMADOC

The user ID of the caller of the procedure must have the EXECUTE privilege on the XSR_ADDSCHEMADOC stored procedure.

XSR_ADDSCHEMADOC syntax diagram

The following syntax diagram shows the CALL statement for invoking XSR_ADDSCHEMADOC.



XSR_ADDSCHEMADOC option descriptions

rschema

An input argument of type VARCHAR(128) that specifies the SQL schema for the XML schema. If a value is specified, it must be SYSXSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name

An input argument of type VARCHAR(128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is "rschema.name". The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure, and XML schema registration cannot yet be

completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation

An input argument of type VARCHAR(1000), which can have a NULL value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the "external name" of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute. The document that references the schemalocation must use valid a URI format.

content

An input parameter of type BLOB(30M) that contains the content of the XML schema document being added. This argument cannot have a NULL value. An XML schema document must be supplied. The content of the XML schema document must be encoded in Unicode.

docproperty

An input parameter of type BLOB(5M) that indicates the properties for the XML schema document being added. This parameter can have a NULL value; otherwise, the value is an XML document.

Example of XSR_ADDSCHEMADOC

The following example calls the XSR_ADDSCHEMADOC stored procedure:

```
CALL SYSPROC.XSR_ADDSCHEMADOC(
  'SYSXSR',
  'POschema',
  'http://myPOschema/PO.xsd',
  :schema_content,
  :schema_properties)
```

In this example, XSR_ADDSCHEMADOC folds the name POschema to uppercase, so the name of the XML schema that is added is POSCHEMA. If you do not want XSR_ADDSCHEMADOC to fold POschema to uppercase, you need to delimit the name with double quotation marks ("), as in the following example.

```
CALL SYSPROC.XSR_ADDSCHEMADOC(
  'SYSXSR',
  '"POschema"',
  'http://myPOschema/PO.xsd',
  :schema_content,
  :schema_properties)
```

 Command line processor (DB2 Commands)

 Example of XML schema registration and removal using stored procedures (DB2 Programming for XML)

XSR_COMPLETE stored procedure

The XSR_COMPLETE procedure is the final stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XSR.

An XML schema is not available for validation until the schema registration completes through a call to this stored procedure.

Environment for XSR_COMPLETE

XSR_COMPLETE requires a WLM-established stored procedures address space that is configured for running Java routines.

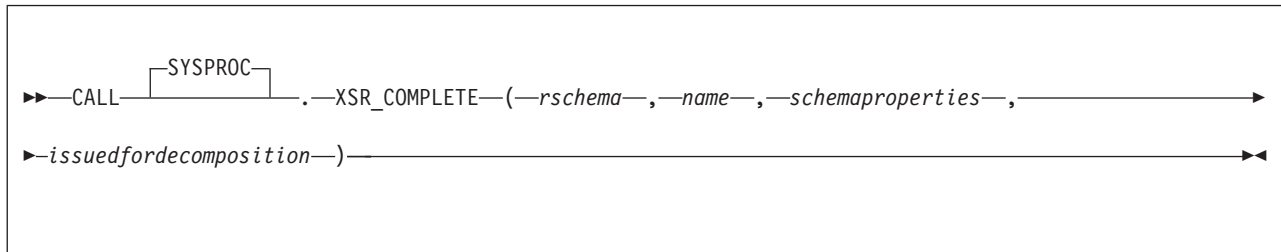
Recommendation: Use WLM environment DSNWLM_JAVA for running XSR_COMPLETE. The startup procedure for this environment, DSNWLMJ, can be configured during installation, using installation panel DSNTIPRH, XML schema processing routines.

Authorization required for XSR_COMPLETE

The user ID of the caller of the procedure must have the EXECUTE privilege on the XSR_COMPLETE stored procedure.

XSR_COMPLETE syntax diagram

The following syntax diagram shows the CALL statement for invoking XSR_COMPLETE.



XSR_COMPLETE option descriptions

rschema

An input argument of type VARCHAR(128) that specifies the SQL schema for the XML schema. If a value is specified, it must be SYSXSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name

An input argument of type VARCHAR(128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemaproperties

An input argument of type BLOB(5M) that specifies properties, if any, associated with the XML schema. The value for this argument is either NULL, if there are no associated properties, or an XML document representing the properties for the XML schema.

issuedfordecomposition

This value must be zero. XML schema decomposition is not supported.

Example of XSR_COMPLETE

The following example calls the XSR_COMPLETE stored procedure:

```
CALL SYSPROC.XSR_COMPLETE(
  'SYSXSR',
  'POschema',
  :schemaproperty_host_var,
  0)
```

In this example, XSR_COMPLETE folds the name POschema to uppercase, so the name of the XML schema for which registration is completed is POSCHEMA. If you do not want XSR_COMPLETE to fold POschema to uppercase, you need to delimit the name with double quotation marks ("), as in the following example.

```
CALL SYSPROC.XSR_COMPLETE(
  'SYSXSR',
  '"POschema"',
  :schemaproperty_host_var,
  0)
```

Related concepts:

[Example of XML schema registration and removal using stored procedures \(DB2 Programming for XML\)](#)

[Command line processor \(DB2 Commands\)](#)

Related tasks:

[Additional steps for enabling the stored procedures and objects for XML schema support \(DB2 Installation and Migration\)](#)

XSR_REMOVE stored procedure

The XSR_REMOVE procedure is used to remove all components of an XML schema. After the XML schema is removed, you can reuse the name of the removed XML schema when you register a new XML schema.

Environment for XSR_REMOVE

XSR_REMOVE runs in a WLM-established stored procedures address space.

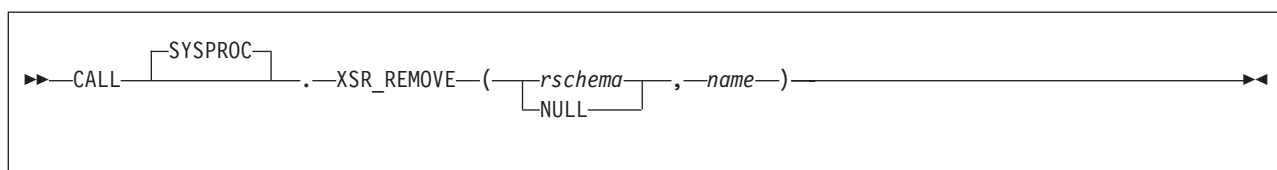
Recommendation: Use WLM environment DSNWLM_XML for running XSR_REMOVE. The startup procedure for this environment, DSNWLMX, can be configured during installation, using installation panel DSNTIPRJ, XML schema processing routines.

Authorization required for XSR_REMOVE

The user ID of the caller of the procedure must have the EXECUTE privilege on the XSR_REMOVE stored procedure.

XSR_REMOVE syntax diagram

The following syntax diagram shows the CALL statement for invoking XSR_REMOVE.



XSR_REMOVE option descriptions

rschema

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. If a value is specified, it must be SYSXSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name

An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is "rschema.name". The XML schema name must already exist as a result of calling the XSR_REGISTER stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

XSR_REMOVE notes

If you run XSR_REMOVE against an XML schema that is part of an XML type modifier for a table column, an error occurs.

Example of XSR_REMOVE

The following example calls the XSR_REMOVE stored procedure:

```
CALL SYSPROC.XSR_REMOVE(  
  'SYSXSR',  
  'POschema')
```


In this example, XSR_REMOVE folds the name POschema to uppercase, so the name of the XML schema that is removed is POSCHEMA. If you do not want XSR_REMOVE to fold POschema to uppercase, you need to delimit the name with double quotation marks ("), as in the following example.

```
CALL SYSPROC.XSR_REMOVE(  
  'SYSXSR',  
  '"POschema"')
```

Related concepts:

 [Command line processor \(DB2 Commands\)](#)

 [Example of XML schema registration and removal using stored procedures \(DB2 Programming for XML\)](#)

 [XML schema validation with an XML type modifier \(DB2 Programming for XML\)](#)

Chapter 15. Coding methods for distributed data

You can access distributed data by using three-part table names or explicit connect statements.

 **Introductory concepts:**

Distributed data (Introduction to DB2 for z/OS)

Effects of distributed data on programming (Introduction to DB2 for z/OS)

Distributed data access (Introduction to DB2 for z/OS)

Three-part table names are described in “Accessing distributed data by using three-part table names.” Explicit connect statements are described in “Accessing distributed data by using explicit CONNECT statements” on page 874.

These two methods of coding applications for distributed access are illustrated by the following example.

Example: Spiffy Computer has a master project table that supplies information about all projects that are currently active throughout the company. Spiffy has several branches in various locations around the world, each a DB2 location that maintains a copy of the project table named DSN8A10.PROJ. The main branch location occasionally inserts data into all copies of the table. The application that makes the inserts uses a table of location names. For each row that is inserted, the application executes an INSERT statement in DSN8A10.PROJ for each location.

Copying a table from a remote location: To copy a table from one location to another, you can either write your own application program or use the DB2 DataPropagator product.

Related concepts:

 Monitoring DB2 in distributed environments (DB2 Performance)

Related tasks:

 Improving performance for applications that access distributed data (DB2 Performance)

Accessing distributed data by using three-part table names

You can use three-part table names to access data at a remote location through DRDA access.

When you use three-part table names, you must create copies of the package that you used at the local site at all possible remote locations that could be accessed by the three-part table name references. You must also explicitly or generically specify remote packages in the PKLIST of the PLAN that is used by the application.

Recommendation: Always use an alias, which resolves to a three-part table name, rather than specifying a specific three-part table name in an SQL statement. Using an alias will permit you to physically move the location of the table as needed. By using an alias, you can drop and re-create the alias by specifying the table's new remote location and then rebind the packages of the application.

In a three-part table name, the first part denotes the location. The local DB2 makes and breaks an implicit connection to a remote server as needed.

When a three-part name is parsed and forwarded to a remote location, any special register settings are automatically propagated to remote server. This allows the SQL statements to process the same way no matter at what site a statement is run.

Example

The following example assumes that all systems involved implement two-phase commit. This example suggests updating several systems in a loop and ending the unit of work by committing only when the loop is complete. Updates are coordinated across the entire set of systems.

Spiffy's application uses a location name to construct a three-part table name in an INSERT statement. It then prepares the statement and executes it dynamically. The values to be inserted are transmitted to the remote location and substituted for the parameter markers in the INSERT statement.

The following overview shows how the application uses aliases for three-part names:

```
Read in the alias values
Do for all locations
    Read location name
    Set up statement to prepare
    Prepare statement
a   Execute statement
End loop
Commit
```

After the application obtains the next alias of a remote table to be inserted, For example, REGION1PROJ (which is the DSN8A10.PROJ table at location SAN_JOSE), it creates the following character string:

```
INSERT INTO REGION1PROJ VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

The alias is created as follows:

```
CREATE ALIAS REGION1PROJ FOR SAN_JOSE.DSN8A10.PROJ
```

The application assigns the character string to the variable INSERTX and then executes these statements:

```
EXEC SQL
    PREPARE STMT1 FROM :INSERTX;
EXEC SQL
    EXECUTE STMT1 USING :PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                       :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ;
```

The host variables for Spiffy's project table match the declaration for the sample project table.

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the INSERT or, if a failure has prevented any location from inserting, all other locations have rolled back the INSERT. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

Three-part names and multiple servers

Recommendation: Always use an asterisk (*) for the location name in a pklst. Never use the explicit location name unless you are sure that no other location could ever be accessed.

The following steps are recommended:

1. Bind the DBRM into a package at the local DB2.
2. Bind package copy at the first target site of the alias.
3. Bind package copy at the target site.

Related concepts:

 Aliases and synonyms (DB2 SQL)

Related tasks:

“Binding packages at a remote location” on page 944

“Including dynamic SQL in your program” on page 193

Related reference:

 Project table (DSN8A10.PROJ) (Introduction to DB2 for z/OS)

Accessing remote declared temporary tables by using three-part table names

You can access a remote declared temporary table by using a three-part name. However, if you combine explicit CONNECT statements and three-part names in your application, a reference to a remote declared temporary table must be a forward reference.

In a CREATE GLOBAL TEMPORARY TABLE or DECLARE GLOBAL TEMPORARY TABLE statement, you cannot specify an alias that resolves to a three-part name object at a remote location. You also cannot specify a three-part name object even if the location of the three-part name refers to the location where the object is being created or declared.

Example

You can perform the following series of actions, which includes a forward reference to a declared temporary table:

```
EXEC SQL CONNECT TO CHICAGO;           /* Connect to the remote site */
EXEC SQL
  DECLARE GLOBAL TEMPORARY TABLE T1 /* Define the temporary table */
  (CHARCOL CHAR(6) NOT NULL)         /* at the remote site */
  ON COMMIT DROP TABLE;
EXEC SQL CONNECT RESET;               /* Connect back to local site */
EXEC SQL INSERT INTO CHICAGO.SESSION.T1
  (VALUES 'ABCDEF');                  /* Access the temporary table*/
                                       /* at the remote site (forward reference) */
```

However, you cannot perform the following series of actions, which includes a backward reference to the declared temporary table:

```
EXEC SQL
  DECLARE GLOBAL TEMPORARY TABLE T1 /* Define the temporary table */
  (CHARCOL CHAR(6) NOT NULL)         /* at the local site (ATLANTA)*/
  ON COMMIT DROP TABLE;
EXEC SQL CONNECT TO CHICAGO;         /* Connect to the remote site */
```

```
EXEC SQL INSERT INTO ATLANTA.SESSION.T1
  (VALUES 'ABCDEF');          /* Cannot access temp table */
                             /* from the remote site (backward reference)*/
```

Example using an alias

You can perform the following series of actions, which includes a forward reference to a declared temporary table using an alias. First you need to declare the alias at the requester. The name you give the alias must resolve to match the real name.

```
CREATE APPLT1 FOR CHICAGO.SESSION.T1
```

The CONNECT and DECLARE statements refer to the real declared temp table.

```
EXEC SQL CONNECT TO CHICAGO;
EXEC SQL DECLARE GLOBAL TEMPORARY TABLE T1
  (CHARCOL CHAR(6) NOT NULL)
  ON COMMIT DROP TABLE;
EXEC SQL CONNECT RESET;
EXEC SQL INSERT INTO APPLT1 VALUES ('ABCDEF');
```

Accessing distributed data by using explicit CONNECT statements

When you use explicit CONNECT statements to access distributed data, the application program explicitly connects to each new server.

You must bind the DBRMs for the SQL statements to be executed at the server to packages that reside at that server.

The following example assumes that all systems involved implement two-phase commit. This example suggests updating several systems in a loop and ending the unit of work by committing only when the loop is complete. Updates are coordinated across the entire set of systems.

In this example, Spiffy's application executes CONNECT for each server in turn, and the server executes INSERT. In this case, the tables to be updated each have the same name, although each table is defined at a different server. The application executes the statements in a loop, with one iteration for each server.

The application connects to each new server by means of a host variable in the CONNECT statement. CONNECT changes the special register CURRENT SERVER to show the location of the new server. The values to insert in the table are transmitted to a location as input host variables.

The following overview shows how the application uses explicit CONNECTs:

```
Read input values
Do for all locations
  Read location name
  Connect to location
  Execute insert statement
End loop
Commit
Release all
```

For example, the application inserts a new location name into the variable LOCATION_NAME and executes the following statements:

```
EXEC SQL
  CONNECT TO :LOCATION_NAME;
EXEC SQL
  INSERT INTO DSN8A10.PROJ VALUES (:PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                                   :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ);
```

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the INSERT or, if a failure has prevented any location from inserting, all other locations have rolled back the INSERT. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

The host variables for Spiffy's project table match the declaration for the sample project table. LOCATION_NAME is a character-string variable of length 16.

Related reference:

[🔗](#) Project table (DSN8A10.PROJ) (Introduction to DB2 for z/OS)

Specifying a location alias name for multiple sites

You can override the location name that an application uses to access a server.

DB2 uses the DBALIAS value in the SYSIBM.LOCATIONS table to override the location name that an application uses to access a server.

For example, suppose that an employee database is deployed across two sites and that both sites make themselves known as location name EMPLOYEE. To access each site, insert a row for each site into SYSIBM.LOCATIONS with the location names SVL_EMPLOYEE and SJ_EMPLOYEE. Both rows contain EMPLOYEE as the DBALIAS value. When an application issues a CONNECT TO SVL_EMPLOYEE statement, DB2 searches the SYSIBM.LOCATIONS table to retrieve the location and network attributes of the database server. Because the DBALIAS value is not blank, DB2 uses the alias EMPLOYEE, and not the location name, to access the database.

If the application uses fully qualified object names in its SQL statements, DB2 sends the statements to the remote server without modification. For example, suppose that the application issues the statement SELECT * FROM SVL_EMPLOYEE.authid.table with the fully-qualified object name. However, DB2 accesses the remote server by using the EMPLOYEE alias. The remote server must identify itself as both SVL_EMPLOYEE and EMPLOYEE; otherwise, it rejects the SQL statement with a message indicating that the database is not found. If the remote server is DB2, the location SVL_EMPLOYEE might be defined as a location alias for EMPLOYEE. DB2 z/OS servers are defined with this alias by using the DDF ALIAS statement of the DSNJU003 change log inventory utility. DB2 locally executes any SQL statements that contain fully qualified object names if the high-level qualifier is the location name or any of its alias names.

Related reference:

[🔗](#) SYSIBM.LOCATIONS table (DB2 SQL)

[🔗](#) DSNJU003 (change log inventory) (DB2 Utilities)

Releasing connections

When you connect to remote locations explicitly, you must also terminate those connections explicitly.

To break the connections, you can use the `RELEASE` statement. The `RELEASE` statement differs from the `CONNECT` statement in the following ways:

- While the `CONNECT` statement makes an immediate connection, the `RELEASE` statement **does not** immediately break a connection. The `RELEASE` statement labels connections for release at the next commit point. A connection that has been labeled for release is in the *release-pending state* and can still be used before the next commit point.
- While the `CONNECT` statement connects to exactly one remote system, you can use the `RELEASE` statement to specify a single connection or a set of connections for release at the next commit point.

By using the `RELEASE` statement, you can place any of the following connections in the release-pending state:

- A specific connection that the next unit of work does not use:
`EXEC SQL RELEASE SPIFFY1;`
- The current SQL connection, whatever its location name:
`EXEC SQL RELEASE CURRENT;`
- All connections except the local connection:
`EXEC SQL RELEASE ALL;`

Transmitting mixed data

Mixed data is data that contains both character and graphic data.

If you transmit mixed data between your local system and a remote system, put the data in varying-length character strings instead of fixed-length character strings.

Converting mixed data: When ASCII MIXED data or Unicode MIXED data is converted to EBCDIC MIXED, the converted string is longer than the source string. An error occurs if that conversion is performed on a fixed-length input host variable. The remedy is to use a varying-length string variable with a maximum length that is sufficient to contain the expansion.

Identifying the server at run time

You can request the location name of the system to which you are connected.

The special register `CURRENT SERVER` contains the location name of the system you are connected to. You can assign that name to a host variable with a statement like this:

```
EXEC SQL SET :CS = CURRENT SERVER;
```

SQL limitations at dissimilar servers

When you execute SQL statements on a remote server that is running another DB2 family product, certain limitations exist. Generally, a program that uses DRDA access can use SQL statements and clauses that are supported by a remote server, even if they are not supported by the local server.

The following examples suggest what to expect from dissimilar servers:

- They support `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `DECLARE CURSOR`, and `FETCH`, but details vary.

Example: DB2 for Linux, UNIX, and Windows and DB2 for i support a form of INSERT that allows for multiple rows of input data. In this case, the VALUES clause is followed by multiple lists in parentheses. Each list represents the values to be inserted for a row of data. DB2 for z/OS does not support this form of INSERT.

- Data definition statements vary more widely.

Example: DB2 for z/OS supports ROWID columns; DB2 for Linux, UNIX, and Windows does not support ROWID columns. Any data definition statements that use ROWID columns cannot run across all platforms.

- Statements can have different limits.

Example: A query in DB2 for z/OS can have 750 columns; for other systems, the maximum is higher. But a query using 750 or fewer columns could execute in all systems.

- Some statements are not sent to the server but are processed completely by the requester. You cannot use those statements in a remote package even though the server supports them.
- In general, if a statement to be executed at a remote server contains host variables, a DB2 requester assumes them to be input host variables unless it supports the syntax of the statement and can determine otherwise. If the assumption is not valid, the server rejects the statement.

Related reference:

 Characteristics of SQL statements in DB2 for z/OS (DB2 SQL)

Support for executing long SQL statements in a distributed environment

A distributed application can send prepared SQL statements exceed 32 KB in size. If the statements exceed 32 KB in size, the server must support these long statements.

If a distributed application assigns an SQL statement to a DBCLOB (UTF-16) variable and sends the prepared statement to a remote server, the remote DB2 server converts it to UTF-8. If the remote server does not support UTF-8, the requester converts the statement to the system EBCDIC CCSID before sending it to the remote server.

Distributed queries against ASCII or Unicode tables

When you perform a distributed query, the server determines the encoding scheme of the result table.

When a distributed query against an ASCII or Unicode table arrives at the DB2 for z/OS server, the server indicates in the reply message that the columns of the result table contain ASCII or Unicode data, rather than EBCDIC data. The reply message also includes the CCSIDs of the data to be returned. The CCSID of data from a column is the CCSID that was in effect when the column was defined.

The encoding scheme in which DB2 returns data depends on two factors:

- The encoding scheme of the requesting system.

If the requester is ASCII or Unicode, the returned data is ASCII or Unicode. If the requester is EBCDIC, the returned data is EBCDIC, even though it is stored

at the server as ASCII or Unicode. However, if the SELECT statement that is used to retrieve the data contains an ORDER BY clause, the data displays in ASCII or Unicode order.

- Whether the application program overrides the CCSID for the returned data. The ways to do this are as follows:

- For static SQL

You can bind a plan or package with the ENCODING bind option to control the CCSIDs for all static data in that plan or package. For example, if you specify ENCODING(UNICODE) when you bind a package at a remote DB2 for z/OS system, the data that is returned in host variables from the remote system is encoded in the default Unicode CCSID for that system.

- For static or dynamic SQL


An application program can specify overriding CCSIDs for individual host variables in DECLARE VARIABLE statements.

An application program that uses an SQLDA can specify an overriding CCSID for the returned data in the SQLDA. When the application program executes a FETCH statement, you receive the data in the CCSID that is specified in the SQLDA.

Related tasks:

“Setting the CCSID for host variables” on page 178

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

Restrictions when using scrollable cursors to access distributed data

The restrictions that exist for scrollable cursors depend on what the requestor and the server support.

If a DB2 for z/OS server processes an OPEN cursor statement for a scrollable cursor, and the OPEN cursor statement comes from a requester that does not support scrollable cursors, the DB2 for z/OS server returns an SQL error. However, if a stored procedure at the server uses a scrollable cursor to return a result set, the down-level requester can access data through that cursor. The DB2 for z/OS server converts the scrollable result set cursor to a non-scrollable cursor. The requester can retrieve the data using sequential FETCH statements.

Restrictions when using rowset-positioned cursors to access distributed data

The restrictions that exist for row-positioned cursors depend on what the requestor and the server support.

If a DB2 for z/OS server processes an OPEN cursor statement for a rowset-positioned cursor, and the OPEN cursor statement comes from a requester that does not support rowset-positioned cursors, the DB2 for z/OS server returns an SQL error. However, if a stored procedure at the server uses a rowset-positioned cursor to return a result set, the down-level requester can access data through that cursor by using row-positioned FETCH statements.

WebSphere MQ with DB2

WebSphere® MQ is a message handling system that enables applications to communicate in a distributed environment across different operating systems and networks.

WebSphere MQ handles the communication from one program to another by using application programming interfaces (APIs). You can use any of the following APIs to interact with the WebSphere MQ message handling system:

- Message Queue Interface (MQI)
- WebSphere MQ classes for Java
- WebSphere MQ classes for Java Message Service (JMS)

DB2 provides its own application programming interface to the WebSphere MQ message handling system through a set of external user-defined functions, which are called DB2 MQ functions. You can use these functions in SQL statements to combine DB2 database access with WebSphere MQ message handling. The DB2 MQ functions use the MQI.

Related reference:

 [WebSphere MQ information center](#)

WebSphere MQ messages

WebSphere MQ uses messages to pass information between applications.

Messages consist of the following parts:

- The message attributes, which identify the message and its properties.
- The message data, which is the application data that is carried in the message.

Related concepts:

“DB2 MQ functions and DB2 MQ XML stored procedures” on page 881

WebSphere MQ message handling

Conceptually, the WebSphere MQ message handling system takes a piece of information (the message) and sends it to its destination. MQ guarantees delivery, despite any network disruptions that might occur.

In WebSphere MQ, a destination is called a message queue, and a queue resides in a queue manager. Applications can put messages on queues or get messages from them.

DB2 communicates with the WebSphere message handling system through a set of external user-defined functions, which are called DB2 MQ functions. These functions use the MQI.

When you send a message, you must specify the following three components:

message data

Defines what is sent from one program to another.

service

Defines where the message is going to or coming from. The parameters for managing a queue are defined in the service, which is typically defined by a system administrator. The complexity of the parameters in the service is hidden from the application program.

policy Defines how the message is handled. Policies control such items as:


- The attributes of the message, for example, the priority.
- Options for send and receive operations, for example, whether an operation is part of a unit of work.

The default service and policy are set as part of defining the WebSphere MQ configuration for a particular installation of DB2. (This action is typically performed by a system administrator.) DB2 provides the default service DB2.DEFAULT.SERVICE and the default policy DB2.DEFAULT.POLICY.

Related tasks:

 Additional steps for enabling WebSphere MQ user-defined functions (DB2 Installation and Migration)

Related reference:

 WebSphere MQ information center

WebSphere MQ message handling with the MQI:

One way to send and receive WebSphere MQ messages from DB2 applications is to use the DB2 MQ functions that use MQI.

These MQI-based functions use the services and policies that are defined in two DB2 tables, SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE. These tables are user-managed and are typically created and maintained by a system administrator. Each table contains a row for the default service and policy that are provided by DB2.

The application program does not need know the details of the services and policies that are defined in these tables. The application need only specify which service and policy to use for each message that it sends and receives. The application specifies this information when it calls a DB2 MQ function.

Related concepts:

“DB2 MQ functions and DB2 MQ XML stored procedures” on page 881

Related reference:

“DB2 MQ tables” on page 884

DB2 MQI services:

A service describes a destination to which an application sends messages or from which an application receives messages. DB2 Message Queue Interface (MQI) services are defined in the DB2 table SYSIBM.MQSERVICE_TABLE.

The MQI-based DB2 MQ functions use the services that are defined in the DB2 table SYSIBM.MQSERVICE_TABLE. This table is user-managed and is typically created and maintained by a system administrator. This table contains a row for each defined service, including your customized services and the default service that is provided by DB2.

The application program does not need know the details of the defined services. When an application program calls an MQI-based DB2 MQ function, the program selects a service from SYSIBM.MQSERVICE_TABLE by specifying it as a parameter.

Related concepts:

“DB2 MQ functions and DB2 MQ XML stored procedures” on page 881

“WebSphere MQ message handling” on page 879

Related reference:

“DB2 MQ tables” on page 884

DB2 MQI policies:

A policy controls how the MQ messages are handled. DB2 Message Queue Interface (MQI) policies are defined in the DB2 table SYSIBM.MQPOLICY_TABLE.

The MQI-based DB2 MQ functions use the policies that are defined in the DB2 table SYSIBM.MQPOLICY_TABLE. This table is user-managed and is typically created and maintained by a system administrator. This table contains a row for each defined policy, including your customized policies and the default policy that is provided by DB2.

The application program does not need know the details of the defined policies. When an application program calls an MQI-based DB2 MQ function, the program selects a policy from SYSIBM.MQPOLICY_TABLE by specifying it as a parameter.

Related concepts:

“DB2 MQ functions and DB2 MQ XML stored procedures”

“WebSphere MQ message handling” on page 879

Related reference:

“DB2 MQ tables” on page 884

DB2 MQ functions and DB2 MQ XML stored procedures

You can use the DB2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

The DB2 MQ functions support the following types of operations:

- Send and forget, where no reply is needed.
- Read or receive, where one or all messages are either read without removing them from the queue, or received and removed from the queue.
- Request and response, where a sending application needs a response to a request.
- Publish and subscribe, where messages are assigned to specific publisher services and are sent to queues. Applications that subscribe to the corresponding subscriber service can monitor specific messages.

You can use the DB2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue. You can send a request to a message queue and receive a response, and you can also publish messages to the WebSphere MQ publisher and subscribe to messages that have been published with specific topics. The DB2 MQ XML functions and stored procedures enable you to query XML documents and then publish the results to a message queue.

The DB2 MQ functions include scalar functions, table functions, and XML-specific functions. For each of these functions, you can call a version that uses the MQI. The function signatures are the same. However, the qualifying schema names are different. To call an MQI-based function, specify the schema name DB2MQ.

Requirement: Before you can call the version of these functions that uses MQI , you need to populate the DB2 MQ tables.

The following table describes the DB2 MQ scalar functions.

Table 138. DB2 MQ scalar functions

Scalar function	Description
MQREAD (<i>receive-service, service-policy</i>)	MQREAD returns a message in a VARCHAR variable from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation does not remove the message from the head of the queue but instead returns it. If no messages are available to be returned, a null value is returned.
MQREADCLOB (<i>receive-service, service-policy</i>)	MQREADCLOB returns a message in a CLOB variable from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation does not remove the message from the head of the queue but instead returns it. If no messages are available to be returned, a null value is returned.
MQRECEIVE (<i>receive-service, service-policy, correlation-id</i>)	MQRECEIVE returns a message in a VARCHAR variable from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation removes the message from the queue. If <i>correlation-id</i> is specified, the first message with a matching correlation identifier is returned; if <i>correlation-id</i> is not specified, the message at the beginning of queue is returned. If no messages are available to be returned, a null value is returned.
MQRECEIVECLOB (<i>receive-service, service-policy, correlation-id</i>)	MQRECEIVECLOB returns a message in a CLOB variable from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation removes the message from the queue. If <i>correlation-id</i> is specified, the first message with a matching correlation identifier is returned; if <i>correlation-id</i> is not specified, the message at the head of queue is returned. If no messages are available to be returned, a null value is returned.
MQSEND (<i>send-service, service-policy, msg-data, correlation-id</i>)	MQSEND sends the data in a VARCHAR or CLOB variable <i>msg-data</i> to the MQ location specified by <i>send-service</i> , using the policy defined in <i>service-policy</i> . An optional user-defined message correlation identifier can be specified by <i>correlation-id</i> . The return value is 1 if successful or 0 if not successful.

Notes:

1. You can send or receive messages in VARCHAR variables or CLOB variables. The maximum length for a message in a VARCHAR variable is 32 KB. The maximum length for a message in a CLOB variable is 2 MB.
- 2.

The following table describes the MQ table functions that DB2 can use.

Table 139. DB2 MQ table functions

Table function	Description
MQREADALL (<i>receive-service, service-policy, num-rows</i>)	MQREADALL returns a table that contains the messages and message metadata in VARCHAR variables from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation does not remove the messages from the queue. If <i>num-rows</i> is specified, a maximum of <i>num-rows</i> messages is returned; if <i>num-rows</i> is not specified, all available messages are returned.
MQREADALLCLOB (<i>receive-service, service-policy, num-rows</i>)	MQREADALLCLOB returns a table that contains the messages and message metadata in CLOB variables from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation does not remove the messages from the queue. If <i>num-rows</i> is specified, a maximum of <i>num-rows</i> messages is returned; if <i>num-rows</i> is not specified, all available messages are returned.

Table 139. DB2 MQ table functions (continued)

Table function	Description
MQRECEIVEALL (<i>receive-service</i> , <i>service-policy</i> , <i>correlation-id</i> , <i>num-rows</i>)	MQRECEIVEALL returns a table that contains the messages and message metadata in VARCHAR variables from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation removes the messages from the queue. If <i>correlation-id</i> is specified, only those messages with a matching correlation identifier are returned; if <i>correlation-id</i> is not specified, all available messages are returned. If <i>num-rows</i> is specified, a maximum of <i>num-rows</i> messages is returned; if <i>num-rows</i> is not specified, all available messages are returned.
MQRECEIVEALLCLOB (<i>receive-service</i> , <i>service-policy</i> , <i>correlation-id</i> , <i>num-rows</i>)	MQRECEIVEALLCLOB returns a table that contains the messages and message metadata in CLOB variables from the MQ location specified by <i>receive-service</i> , using the policy defined in <i>service-policy</i> . This operation removes the messages from the queue. If <i>correlation-id</i> is specified, only those messages with a matching correlation identifier are returned; if <i>correlation-id</i> is not specified, all available messages are returned. If <i>num-rows</i> is specified, a maximum of <i>num-rows</i> messages is returned; if <i>num-rows</i> is not specified, all available messages are returned.

Notes:

1. You can send or receive messages in VARCHAR variables or CLOB variables. The maximum length for a message in a VARCHAR variable is 32 KB. The maximum length for a message in a CLOB variable is 2 MB.
2. The first column of the result table of a DB2 MQ table function contains the message.
- 3.

Related concepts:

“DB2-supplied stored procedures” on page 811

Related tasks:

[➤](#) Additional steps for enabling WebSphere MQ user-defined functions (DB2 Installation and Migration)

Related reference:

[➤](#) MQREADALL (DB2 SQL)

[➤](#) MQREADALLCLOB (DB2 SQL)

[➤](#) MQRECEIVEALL (DB2 SQL)

[➤](#) MQRECEIVEALLCLOB (DB2 SQL)

[➤](#) WebSphere MQ information center

Generating XML documents from existing tables and sending them to an MQ message queue

You can send data from a DB2 table to the MQ message queue. First put the data in an XML document and then send that document to the message queue.

To generate XML documents from existing tables and send them to an MQ message queue:

1. Compose an XML document by using the DB2 XML publishing functions.
2. Cast the XML document to type VARCHAR or CLOB.
3. Send the document to an MQ message queue by using the appropriate DB2 MQ function.

Related concepts:

“DB2 MQ functions and DB2 MQ XML stored procedures” on page 881

Shredding XML documents from an MQ message queue

When you retrieve XML data from an MQ message queue, you can shred that data into DB2 tables for easy retrievability.

To shred XML documents from an MQ message queue:

1. Retrieve the XML document from an MQ message queue by using the appropriate MQ function.
2. Shred the retrieved message to DB2 tables by using the XML decomposition stored procedure (XDBDECOMPXML).

Related concepts:

“DB2 MQ functions and DB2 MQ XML stored procedures” on page 881

DB2 MQ tables

The DB2 MQ tables contain service and policy definitions that are used by the Message Queue Interface (MQI) based DB2 MQ functions. You must populate the DB2 MQ tables before you can use these MQI-based functions.

The DB2 MQ tables are SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE. These tables are user-managed. You need to create them during the installation or migration process. Installation job DSNTIJRT creates these tables with one default row in each table.

If you previously used the AMI-based DB2 MQ functions, you used AMI configuration files instead of these tables. To use the MQI-based DB2 MQ functions, you need to move the data from those configuration files to the DB2 tables SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE .

The following table describes the columns for SYSIBM.MQSERVICE_TABLE.

Table 140. SYSIBM.MQSERVICE_TABLE column descriptions

Column name	Description
SERVICENAME	This column contains the service name, which is an optional input parameter of the MQ functions. This column is the primary key for the SYSIBM.MQSERVICE_TABLE table.
QUEUEMANAGER	This column contains the name of the queue manager where the MQ functions are to establish a connection.
INPUTQUEUE	This column contains the name of the queue from which the MQ functions are to send and retrieve messages.
CODEDCHARSETID	This column contains the character set identifier for character data in the messages that are sent and received by the MQ functions. This column corresponds to the CodedCharSetId field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the CodedCharSetId field. The default value for this column is 0, which sets the CodedCharSetId field of the MQMD to the value MQCCSI_Q_MGR.

Table 140. SYSIBM.MQSERVICE_TABLE column descriptions (continued)

Column name	Description
ENCODING	<p>This column contains the encoding value for the numeric data in the messages that are sent and received by the MQ functions.</p> <p>This column corresponds to the Encoding field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Encoding field.</p> <p>The default value for this column is 0, which sets the Encoding field in the MQMD to the value MQENC_NATIVE.</p>
DESCRIPTION	This column contains the description of the service.

The following table describes the columns for SYSIBM.MQPOLICY_TABLE.

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions

Column name	Description
POLICYNAME	<p>This column contains the policy name, which is an optional input parameter of the MQ functions.</p> <p>This column is the primary key for the SYSIBM.MQPOLICY_TABLE table.</p>
SEND_PRIORITY	<p>This column contains the priority of the message.</p> <p>This column corresponds to the Priority field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Priority field.</p> <p>The default value for this column is -1, which sets the Priority field in the MQMD to the value MQQPRI_PRIORITY_AS_Q_DEF.</p>
SEND_PERSISTENCE	<p>This column indicates whether the message persists despite any system failures or instances of restarting the queue manager.</p> <p>This column corresponds to the Persistence field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Persistence field.</p> <p>This column can have the following values:</p> <ul style="list-style-type: none"> Q Sets the Persistence field in the MQMD to the value MQPER_PERSISTENCE_AS_Q_DEF. This value is the default. Y Sets the Persistence field in the MQMD to the value MQPER_PERSISTENT. N Sets the Persistence field in the MQMD to the value MQPER_NOT_PERSISTENT.

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_EXPIRY	<p>This column contains the message expiration time, in tenths of a second.</p> <p>This column corresponds to the Expiry field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Expiry field.</p> <p>The default value is -1, which sets the Expiry field to the value MQEI_UNLIMITED.</p>
SEND_RETRY_COUNT	<p>This column contains the number of times that the MQ function is to try to send a message if the procedure fails.</p> <p>The default value is 5.</p>
SEND_RETRY_INTERVAL	<p>This column contains the interval, in milliseconds, between each attempt to send a message.</p> <p>The default value is 1000.</p>
SEND_NEW_CORRELID	<p>This column specifies how the correlation identifier is to be set if a correlation identifier is not passed as an input parameter in the MQ function. The correlation identifier is set in the CorrelId field in the message descriptor structure (MQMD).</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Sets the CorrelId field in the MQMD to binary zeros. This value is the default. Y Specifies that the queue manager is to generate a new correlation identifier and set the CorrelId field in the MQMD to that value. This 'Y' value is equivalent to setting the MQPMO_NEW_CORREL_ID option in the Options field in the put message options structure (MQPMO).
SEND_RESPONSE_MSGID	<p>This column specifies how the MsgId field in the message descriptor structure (MQMD) is to be set for report and reply messages.</p> <p>This column corresponds to the Report field in the MQMD. MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Sets the MQRO_NEW_MSG_ID option in the Report field in the MQMD. This value is the default. P Sets the MQRO_PASS_MSG_ID option in the Report field in the MQMD.

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_RESPONSE_CORRELID	<p>This column specifies how the CorrelID field in the message descriptor structure (MQMD) is to be set for report and reply messages.</p> <p>This column corresponds to the Report field in the MQMD. MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> C Sets the MQRO_COPY_MSG_ID_TO_CORREL_ID option in the Report field in the MQMD. This value is the default. P Sets the MQRO_PASS_CORREL_ID option in the Report field in the MQMD.
SEND_EXCEPTION_ACTION	<p>This column specifies what to do with the original message when it cannot be delivered to the destination queue.</p> <p>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> Q Sets the MQRO_DEAD_LETTER_Q option in the Report field in the MQMD. This value is the default. D Sets the MQRO_DISCARD_MSG option in the Report field in the MQMD. P Sets the MQRO_PASS_DISCARD_AND_EXPIRY option in the Report field in the MQMD.
SEND_REPORT_EXCEPTION	<p>This column specifies whether an exception report message is to be generated when a message cannot be delivered to the specified destination queue and if so, what that report message should contain.</p> <p>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Specifies that an exception report message is not to be generated. No options in the Report field are set. This value is the default. E Sets the MQRO_EXCEPTION option in the Report field in the MQMD. D Sets the MQRO_EXCEPTION_WITH_DATA option in the Report field in the MQMD. F Sets the MQRO_EXCEPTION_WITH_FULL_DATA option in the Report field in the MQMD.

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_REPORT_COA	<p>This column specifies whether the queue manager is to send a confirm-on-arrival (COA) report message when the message is placed in the destination queue, and if so, what that COA message is to contain.</p> <p>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Specifies that a COA message is not to be sent. No options in the Report field are set. This value is the default C Sets the MQRO_COA option in the Report field in the MQMD D Sets the MQRO_COA_WITH_DATA option in the Report field in the MQMD. F Sets the MQRO_COA_WITH_FULL_DATA option in the Report field in the MQMD.
SEND_REPORT_COD	<p>This column specifies whether the queue manager is to send a confirm-on-delivery (COD) report message when an application retrieves and deletes a message from the destination queue, and if so, what that COD message is to contain.</p> <p>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Specifies that a COD message is not to be sent. No options in the Report field are set. This value is the default. C Sets the MQRO_COD option in the Report field in the MQMD. D Sets the MQRO_COD_WITH_DATA option in the Report field in the MQMD. F Sets the MQRO_COD_WITH_FULL_DATA option in the Report field in the MQMD.

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_REPORT_EXPIRY	<p>This column specifies whether the queue manager is to send an expiration report message if a message is discarded before it is delivered to an application, and if so, what that message is to contain.</p> <p>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Specifies that an expiration report message is not to be sent. No options in the Report field are set. This value is the default. C Sets the MQRO_EXPIRATION option in the Report field in the MQMD. D Sets the MQRO_EXPIRATION_WITH_DATA option in the Report field in the MQMD. F Sets the MQRO_EXPIRATION_WITH_FULL_DATA option in the Report field in the MQMD.
SEND_REPORT_ACTION	<p>This column specifies whether the receiving application sends a positive action notification (PAN), a negative action notification (NAN), or both.</p> <p>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</p> <p>This column can have one of the following values:</p> <ul style="list-style-type: none"> N Specifies that neither notification is to be sent. No options in the Report field are set. This value is the default. P Sets the MQRO_PAN option in the Report field in the MQMD. T Sets the MQRO_NAN option in the Report field in the MQMD. B Sets both the MQRO_PAN and MQRO_NAN options in the Report field in the MQMD.

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SEND_MSG_TYPE	<p>This column contains the type of message.</p> <p>This column corresponds to the MsgType field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the MsgType field.</p> <p>This column can have one of the following values:</p> <p>DTG Sets the MsgType field in the MQMD to MQMT_DATAGRAM. This value is the default.</p> <p>REQ Sets the MsgType field in the MQMD to MQMT_REQUEST.</p> <p>RLY Sets the MsgType field in the MQMD to MQMT_REPLY.</p> <p>RPT Sets the MsgType field in the MQMD to MQMT_REPORT.</p>
REPLY_TO_Q	<p>This column contains the name of the message queue to which the application that issued the MQGET call is to send reply and report messages.</p> <p>This column corresponds to the ReplyToQ field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the ReplyToQ field.</p> <p>The default value for this column is SAME AS INPUT_Q, which sets the name to the queue name that is defined in the service that was used for sending the message. If no service was specified, the name is set to DB2MQ_DEFAULT_Q, which is the name of the input queue for the default service.</p>
REPLY_TO_QMGR	<p>This column contains the name of the queue manager to which the reply and report messages are to be sent.</p> <p>This column corresponds to the ReplyToQMgr field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the ReplyToQMgr field.</p> <p>The default value for this column is SAME AS INPUT_QMGR, which sets the name to the queue manager name that is defined in the service that was used for sending the message. If no service was specified, the name is set to the name of the queue manager for the default service.</p>
RCV_WAIT_INTERVAL	<p>This column contains the time, in milliseconds, that DB2 is to wait for messages to arrive in the queue.</p> <p>This column corresponds to the WaitInterval field in the get message options structure (MQGMO). MQ functions use the value in this column to set the WaitInterval field.</p> <p>The default is 10.</p>

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
RCV_CONVERT	<p>This column indicates whether to convert the application data in the message to conform to the CodedCharSetId and Encoding values of the specified MQ service.</p> <p>This column corresponds to the Options field in the get message options structure (MQGMO). MQ functions use the value in this column to set the Options field.</p> <p>This column can have one of the following values:</p> <p>Y Sets the MQGMO_CONVERT option in the Options field in the MQGMO. This value is the default.</p> <p>N Specifies that no data is to be converted.</p>
RCV_ACCEPT_TRUNC_MSG	<p>This column specifies the behavior of the MQ function when oversized messages are retrieved.</p> <p>This column corresponds to the Options field in the get message options structure (MQGMO). MQ functions use the value in this column to set the Options field.</p> <p>This column can have one of the following values:</p> <p>Y Sets the MQGMO_ACCEPT_TRUNCATED_MSG option in the Options field in the MQGMO. This value is the default.</p> <p>N Specifies that no messages are to be truncated. If the message is too large to fit in the buffer, the MQ function terminates with an error.</p> <p>Recommendation: Set this column to Y. In this case, if the message buffer is too small to hold the complete message, the MQ function can fill the buffer with as much of the message as the buffer can hold.</p>
REV_OPEN_SHARED	<p>This column specifies the input queue mode when messages are retrieved.</p> <p>This column corresponds to the Options parameter for an MQOPEN call. MQ functions use the value in this column to set the Options parameter.</p> <p>This column can have one of the following values:</p> <p>S Sets the MQOO_INPUT_SHARED option. This value is the default.</p> <p>E Sets the MQ option MQOO_INPUT_EXCLUSIVE option.</p> <p>D Sets the MQ option MQOO_INPUT_AS_Q_DEF option.</p>

Table 141. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

Column name	Description
SYNCPOINT	<p>This column indicates whether the MQ function is to operate within the protocol for a normal unit of work.</p> <p>This column can have one of the following values:</p> <p>Y Specifies that the MQ function is to operate within the protocol for a normal unit of work. Use this value for two-phase commit environments. This value is the default.</p> <p>N Specifies that the MQ function is to operate outside the protocol for a normal unit of work. Use this value for one-phase commit environments.</p>
DESC	This column contains the description of the policy.

Related reference:

[➤ Core WLM environments for DB2-supplied routines \(DB2 Installation and Migration\)](#)

[➤ WebSphere MQ information center](#)

Basic messaging with WebSphere MQ

The most basic form of messaging with the DB2 MQ functions occurs when all database applications connect to the same DB2 database server. Clients can be local to the database server or distributed in a network environment.

In a simple scenario, client A invokes the MQSEND function to send a user-defined string to the location that is defined by the default service. DB2 executes the MQ functions that perform this operation on the database server. At some later time, client B invokes the MQRECEIVE function to remove the message at the head of the queue that is defined by the default service, and return it to the client. DB2 executes the MQ functions that perform this operation on the database server.

Database clients can use simple messaging in a number of ways:

- Data collection

Information is received in the form of messages from one or more sources. An information source can be any application. The data is received from queues and stored in database tables for additional processing.
- Workload distribution

Work requests are posted to a queue that is shared by multiple instances of the same application. When an application instance is ready to perform some work, it receives a message that contains a work request from the head of the queue. Multiple instances of the application can share the workload that is represented by a single queue of pooled requests.
- Application signaling

In a situation where several processes collaborate, messages are often used to coordinate their efforts. These messages might contain commands or requests for work that is to be performed. For more information about this technique, see “Application to application connectivity with WebSphere MQ” on page 895.

The following scenario extends basic messaging to incorporate remote messaging. Assume that machine A sends a message to machine B.

1. The DB2 client executes an MQSEND function call, specifying a target service that has been defined to be a remote queue on machine B.
2. The MQ functions perform the work to send the message. The WebSphere MQ server on machine A accepts the message and guarantees that it will deliver it to the destination that is defined by the service and the current MQ configuration of machine A. The server determines that the destination is a queue on machine B. The server then attempts to deliver the message to the WebSphere MQ server on machine B, trying again as needed.
3. The WebSphere MQ server on machine B accepts the message from the server on machine A and places it in the destination queue on machine B.
4. A WebSphere MQ client on machine B requests the message at the head of the queue.

Sending messages with WebSphere MQ

When you send messages with WebSphere MQ, you choose what data to send, where to send it and when to send it. This type of messaging is called *send and forget*; the sender sends a message and relies on WebSphere MQ to ensure that the message reaches its destination.

To send messages with WebSphere MQ, use MQSEND.

If you send more than one column of information, separate the columns with the characters `|| ' ' ||`.

Example: MQSEND (LASTNAME || ' ' || FIRSTNAME)

The following examples use the DB2MQ schema for two-phase commit, with the default service DB2.DEFAULT.SERVICE and the default policy DB2.DEFAULT.POLICY.

Example: The following SQL SELECT statement sends a message that consists of the string "Testing msg":

```
SELECT DB2MQ.MQSEND ('Testing msg')
FROM SYSIBM.SYSDUMMY1;
COMMIT;
```

The MQSEND function is invoked once because SYSIBM.SYSDUMMY1 has only one row. Because this MQSEND function uses two-phase commit, the COMMIT statement ensures that the message is added to the queue.

When you use single-phase commit, you do not need to use a COMMIT statement. For example:

```
SELECT DB2MQ.MQSEND ('Testing msg')
FROM SYSIBM.SYSDUMMY1;
```

The MQ operation causes the message to be added to the queue.

Example: Assume that you have an EMPLOYEE table, with VARCHAR columns LASTNAME, FIRSTNAME, and DEPARTMENT. To send a message that contains this information for each employee in DEPARTMENT 5LGA, issue the following SQL SELECT statement:

```
SELECT DB2MQ.MQSEND (LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT)
FROM EMPLOYEE WHERE DEPARTMENT = '5LGA';
COMMIT;
```

Message content can be any combination of SQL statements, expressions, functions, and user-specified data. Because this MQSEND function uses two-phase commit, the COMMIT statement ensures that the message is added to the MQ queue.

Related reference:

 MQSEND (DB2 SQL)

Retrieving messages with WebSphere MQ

With WebSphere MQ, programs can read or receive messages. Both reading and receiving operations return the message at the start of the queue. However, the reading operation does not remove the message from the queue, whereas the receiving operation does.

A message that is retrieved using a receive operation can be retrieved only once, whereas a message that is retrieved using a read operation allows the same message to be retrieved many times.

The following examples use the DB2MQ2N schema for two-phase commit, with the default service DB2.DEFAULT.SERVICE and the default policy DB2.DEFAULT.POLICY.

Example: The following SQL SELECT statement reads the message at the head of the queue that is specified by the default service and policy:

```
SELECT DB2MQ2N.MQREAD()  
FROM SYSIBM.SYSDUMMY1;
```

The MQREAD function is invoked once because SYSIBM.SYSDUMMY1 has only one row. The SELECT statement returns a VARCHAR(4000) string. If no messages are available to be read, a null value is returned. Because MQREAD does not change the queue, you do not need to use a COMMIT statement.

Example: The following SQL SELECT statement causes the contents of a queue to be materialized as a DB2 table:

```
SELECT T.*  
FROM TABLE(DB2MQ2N.MQREADALL()) T;
```

The result table T of the table function consists of all the messages in the queue, which is defined by the default service, and the metadata about those messages. The first column of the materialized result table is the message itself, and the remaining columns contain the metadata. The SELECT statement returns both the messages and the metadata.

To return only the messages, issue the following statement:

```
SELECT T.MSG  
FROM TABLE(DB2MQ2N.MQREADALL()) T;
```

The result table T of the table function consists of all the messages in the queue, which is defined by the default service, and the metadata about those messages. This SELECT statement returns only the messages.

Example: The following SQL SELECT statement receives (removes) the message at the head of the queue:

```
SELECT DB2MQ2N.MQRECEIVE()  
FROM SYSIBM.SYSDUMMY1;  
COMMIT;
```


The MQRECEIVE function is invoked once because SYSIBM.SYSDUMMY1 has only one row. The SELECT statement returns a VARCHAR(4000) string. Because this MQRECEIVE function uses two-phase commit, the COMMIT statement ensures that the message is removed from the queue. If no messages are available to be retrieved, a null value is returned, and the queue does not change.

Example: Assume that you have a MESSAGES table with a single VARCHAR(2000) column. The following SQL INSERT statement inserts all of the messages from the default service queue into the MESSAGES table in your DB2 database:

```
INSERT INTO MESSAGES
  SELECT T.MSG
  FROM TABLE(DB2MQ2N.MQRECEIVEALL()) T;
COMMIT;
```

The result table T of the table function consists of all the messages in the default service queue and the metadata about those messages. The SELECT statement returns only the messages. The INSERT statement stores the messages into a table in your database.

Application to application connectivity with WebSphere MQ

Application-to-application connectivity is typically used when putting together a diverse set of application subsystems. To facilitate application integration, WebSphere MQ provides the means to interconnect applications.

The following two scenarios are very common when interconnecting applications:

- *Request-and-reply* communication method
- *Publish-and-subscribe* method

Request-and-reply communication method:

The request-and-reply method enables one application to request the services of another application. One way to do this is for the requester to send a message to the service provider to request that some work be performed. When the work has been completed, the provider might decide to send results, or just a confirmation of completion, back to the requester. Unless the requester waits for a reply before continuing, WebSphere MQ must provide a way to associate the reply with its request.

WebSphere MQ provides a correlation identifier to correlate messages in an exchange between a requester and a provider. The requester marks a message with a known correlation identifier. The provider marks its reply with the same correlation identifier. To retrieve the associated reply, the requester provides that correlation identifier when receiving messages from the queue. The first message with a matching correlation identifier is returned to the requester.

The following examples use the DB2MQ schema for single-phase commit.

Example: The following SQL SELECT statement sends a message consisting of the string "Msg with corr id" to the service MYSERVICE, using the policy MYPOLICY with correlation identifier CORRID1:

```
SELECT DB2MQ.MQSEND ('MYSERVICE', 'MYPOLICY', 'Msg with corr id', 'CORRID1')
  FROM SYSIBM.SYSDUMMY1;
```

The MQSEND function is invoked once because SYSIBM.SYSDUMMY1 has only one row. Because this MQSEND uses single-phase commit, WebSphere MQ adds the message to the queue, and you do not need to use a COMMIT statement.

Example: The following SQL SELECT statement receives the first message that matches the identifier CORRID1 from the queue that is specified by the service MYSERVICE, using the policy MYPOLICY:

```
| SELECT DB2MQ.MQRECEIVE ('MYSERVICE', 'MYPOLICY', 'CORRID1')  
| FROM SYSIBM.SYSDUMMY1;
```

The SELECT statement returns a VARCHAR(4000) string. If no messages are available with this correlation identifier, a null value is returned, and the queue does not change.

Publish-and-subscribe method:

Another common method of application integration is for one application to notify other applications about events of interest. An application can do this by sending a message to a queue that is monitored by other applications. The message can contain a user-defined string or can be composed from database columns.

Simple data publication: In many cases, only a simple message needs to be sent using the MQSEND function. When a message needs to be sent to multiple recipients concurrently, the distribution list facility of the MQSeries® AMI can be used.

You define distribution lists by using the AMI administration tool. A *distribution list* comprises a list of individual services. A message that is sent to a distribution list is forwarded to every service defined within the list. Publishing messages to a distribution list is especially useful when there are multiple services that are interested in every message.

Example: The following example shows how to send a message to the distribution list "InterestedParties":

```
| SELECT DB2MQ.MQSEND ('InterestedParties','Information of general interest')  
| FROM SYSIBM.SYSDUMMY1;
```

To receive published messages, you must first register your application's interest in messages of a given topic and indicate the name of the subscriber service to which messages are sent. An AMI subscriber service defines a broker service and a receiver service. The *broker service* is how the subscriber communicates with the publish-and-subscribe broker. The *receiver service* is the location where messages that match the subscription request are sent.

Example: The following statement subscribes to the topic ALL_EMP and indicates that messages be sent to the subscriber service, "aSubscriber":

```
| SELECT DB2MQ.MQSUBSCRIBE ('aSubscriber','ALL_EMP')  
| FROM SYSIBM.SYSDUMMY1;
```

When an application is subscribed, messages published with the topic, ALL_EMP, are forwarded to the receiver service that is defined by the subscriber service. An application can have multiple concurrent subscriptions. Messages that match the subscription topic can be retrieved by using any of the standard message retrieval functions.

Example: The following statement non-destructively reads the first message, where the subscriber service, "aSubscriber", defines the receiver service as "aSubscriberReceiver":

```
| SELECT DB2MQ.MQREAD ('aSubscriberReceiver')  
| FROM SYSIBM.SYSDUMMY1;
```

To display both the messages and the topics with which they are published, you can use one of the table functions.

Example: The following statement receives the first five messages from "aSubscriberReceiver" and display both the message and the topic for each of the five messages:

```
| SELECT t.msg, t.topic  
| FROM table (DB2MQ.MQRECEIVEALL ('aSubscriberReceiver',5)) t;
```

Example: To read all of the messages with the topic ALL_EMP, issue the following statement:

```
| SELECT t.msg  
| FROM table (DB2MQ.MQREADALL ('aSubscriberReceiver')) t  
| WHERE t.topic = 'ALL_EMP';
```

Note: If you use MQRECEIVEALL with a constraint, your application receives the entire queue, not just those messages that are published with the topic ALL_EMP. This is because the table function is performed before the constraint is applied.

When you are no longer interested in having your application subscribe to a particular topic, you must explicitly unsubscribe.

Example: The following statement unsubscribes from the ALL_EMP topic of the "aSubscriber" subscriber service:

```
| SELECT DB2MQ.MQUNSUBSCRIBE ('aSubscriber', 'ALL_EMP')  
| FROM SYSIBM.SYSDUMMY1;
```

After you issue the preceding statement, the publish-and-subscribe broker no longer delivers messages that match the ALL_EMP topic to the "aSubscriber" subscriber service.

Automated Publication: Another important method in application message publishing is automated publication. Using the trigger facility within DB2 for z/OS, you can automatically publish messages as part of a trigger invocation. Although other techniques exist for automated message publication, the trigger-based approach gives you more freedom in constructing the message content and more flexibility in defining the actions of a trigger. As with the use of any trigger, you must be aware of the frequency and cost of execution.

Example: The following example shows how you can use the MQSeries functions of DB2 for z/OS with a trigger to publish a message each time a new employee is hired:

```
| CREATE TRIGGER new_employee AFTER INSERT ON DSN8A10.EMP  
| REFERENCING NEW AS n  
| FOR EACH ROW MODE DB2SQL  
| SELECT DB2MQ.MQPUBLISH ('HR_INFO_PUB', current date || ' ' ||  
| LASTNAME || ' ' || DEPARTMENT, 'NEW_EMP');
```

Any users or applications that subscribe to the HR_INFO_PUB service with a registered interest in the NEW_EMP topic will receive a message that contains the date, the name, and the department of each new employee when rows are inserted into the DSN8A10.EMP table.

Asynchronous messaging in DB2 for z/OS

Programs can communicate with each other by sending data in messages rather than using constructs like synchronous remote procedure calls. With asynchronous messaging, the program that sends the message proceeds with its processing after sending the message, without waiting for a reply.

If the program needs information from the reply, the program suspends processing and waits for a reply message. If the messaging programs use an intermediate queue that holds messages, the requestor program and the receiver program do not need to be running at the same time. The requestor program places a request message on a queue and then exits. The receiver program retrieves the request from the queue and processes the request.

Asynchronous operations require that the service provider is capable of accepting requests from clients without notice. An asynchronous listener is a program that monitors message transporters, such as WebSphere MQ, and performs actions based on the message type. An asynchronous listener can use WebSphere MQ to receive all messages that are sent to an endpoint. An asynchronous listener can also register a subscription with a publish or subscribe infrastructure to restrict the messages that are received to messages that satisfy specified constraints.

Examples: The following examples show some common uses of asynchronous messaging:

Message accumulator

You can accumulate the messages that are sent asynchronously so that the listener checks for messages and stores those messages automatically in a database. This database, which acts as a message accumulator, can save all messages for a particular endpoint, such as an audit trail. The asynchronous listener can subscribe to a subset of messages, such as *save only high value stock trades*. The message accumulator stores entire messages, and does not provide for selection, transformation, or mapping of message contents to database structures. The message accumulator does not reply to messages.

Message event handler

The asynchronous event handler listens for messages and invokes the appropriate handler (such as a stored procedure) for the message endpoint. You can call any arbitrary stored procedure. The asynchronous listener lets you select, map, or reformat message contents for insertion into one or more database structures.

Asynchronous messaging has the following benefits:

- The client and database do not need to be available at the same time. If the client is available intermittently, or if the client fails between the time the request is issued and the response is sent, it is still possible for the client to receive the reply. Or, if the client is on a mobile computer and becomes disconnected from the database, and if a response is sent, the client can still receive the reply.
- The content of the messages in the database contain information about when to process particular requests. The messages in the database use priorities and the request contents to determine how to schedule the requests.

- An asynchronous message listener can delegate a request to a different node. It can forward the request to a second computer to complete the processing. When the request is complete, the second computer returns a response directly to the endpoint that is specified in the message.
- An asynchronous listener can respond to a message from a supplied client, or from a user-defined application. The number of environments that can act as a database client is greatly expanded. Clients such as factory automation equipment, pervasive devices, or embedded controllers can communicate with DB2 either directly through WebSphere MQ or through some gateway that supports WebSphere MQ.

MQListener in DB2 for z/OS

DB2 for z/OS provides an asynchronous listener, MQListener. MQListener is a framework for tasks that read from WebSphere MQ queues and call DB2 stored procedures with messages as those messages arrive.

MQListener combines messaging with database operations. You can configure the MQListener daemon to listen to the WebSphere MQ message queues that you specify in a configuration database. MQListener reads the messages that arrive from the queue and calls DB2 stored procedures using the messages as input parameters. If the message requires a reply, MQListener creates a reply from the output that is generated by the stored procedure. The message retrieval order is fixed at the highest priority first, and then within each priority the first message received is the first message served.

MQListener runs as a single multi-threaded process on z/OS UNIX System Services. Each thread or task establishes a connection to its configured message queue for input. Each task also connects to a DB2 database on which to run the stored procedure. The information about the queue and the stored procedure is stored in a table in the configuration database. The combination of the queue and the stored procedure is a task.

MQListener tasks are grouped together into named configurations. By default, the configuration name is empty. If you do not specify the name of a configuration for a task, MQListener uses the configuration with an empty name.

Transaction support: There is support for both one-phase and two-phase commit environments. A one-phase commit environment is where DB interactions and MQ interactions are independent. A two-phase commit environment is where DB interactions and MQ interactions are combined in a single unit of work.

'db2mqIn1' is the name of the executable for one phase and 'db2mqIn2' is the name of the executable for two phase.

Logical ordering of messages: The two-phase commit version of the MQListener stored procedure processes messages that are in a group in logical order. The single-phase commit version of the MQListener stored procedure processes messages that are in a group in physical order.

Stored Procedure Interface: The stored procedure interface for MQListener takes the incoming message as input and returns the reply, which might be NULL, as output:

```
schema.proc(in inMsg inMsgType, out outMsg outMsgType)
```

The data type for *inMsgType* and the data type for *outMsgType* can be VARCHAR, VARBINARY, CLOB, or BLOB of any length and are determined at startup. The

input data type and output data type can be different data types. If an incoming message is a request and has a specified reply-to queue, the message in outMsg will be sent to the specified queue. The incoming message can be one of the following message types:

- Datagram
- Datagram with report requested
- Request message with reply
- Request message with reply and report requested

Configuring MQListener in DB2 for z/OS:

Before you can use MQListener, you must configure your database environment so that your applications can use messaging with database operations. You must also configure WebSphere MQ for MQListener.

Use the following procedure to configure the environment for MQListener and to develop a simple application that receives a message, inserts the message in a table, and creates a simple response message:

1. Configure MQListener to run in the DB2 environment.
2. Configure WebSphere MQ for MQListener.
3. Configure MQListener task.
4. Create the sample stored procedure to work with MQListener.
5. Run a simple MQListener application.

Configuring MQListener to run in the DB2 environment:

Configure your database environment so that your applications can use messaging with database operations.

Customize and run installation job DSNTIJML, which is located in *prefix.SDSNSAMP* data set. The job will do the following tasks:

1. Untar and create the necessary files and libraries in z/OS UNIX System Services under the path where MQListener is installed.
2. Create the MQListener configuration table (SYSXML.LISTENERS) in the default database DSNDB04.
3. Bind the DBRM's to the plan DB2MQLSN.

The samples DSNTJML, DSNTJSP and DSNTIJML are located in *prefix.SDSNSAMP* data set.

Ensure that the person who runs the installation job has required authority to create the configuration table and to bind the DBRM's.

Follow the instructions in the README file that is created in the MQListener installation path in z/OS UNIX System Services to complete the configuration process.

Configuring WebSphere MQ for MQListener:

You can run a simple MQListener application with a simple WebSphere MQ configuration. More complex applications might need a more complex configuration. Configure at least two kinds of WebSphere MQ entities: the queue

manager and some local queues. Configure these entities for use in such instances as transaction management, deadletter queue, backout queue, and backout retry threshold.

To configure WebSphere MQ for a simple MQListener application, complete the following steps:

1. Create MQSeries QueueManager. Define the MQSeries subsystem to z/OS and then issue the following command from a z/OS console to start the queue manager:

```
<command-prefix-string> START QMGR
```


command-prefix-string is the command prefix for the MQSeries subsystem.
2. Create Queues under MQSeries QueueManager:

In a simple MQListener application, you typically use the following WebSphere MQ queues:

Deadletter queue

The deadletter queue in WebSphere MQ holds messages that cannot be processed. MQListener uses this queue to hold replies that cannot be delivered, for example, because the queue to which the replies should be sent is full. A deadletter queue is useful in any MQ installation especially for recovering messages that are not sent.

Backout queue

For MQListener tasks that use two-phase commit, the backout queue serves a similar purpose as the deadletter queue. MQListener places the original request in the backout queue after the request is rolled back a specified number of times (called the backout threshold).

Administration queue

The administration queue is used for routing control messages, such as shutdown and restart, to MQListener. If you do not supply an administration queue, the only way to shut down MQListener is to issue a kill command.

Application input and output queues

The application uses input queues and output queues. The application receives messages from the input queue and sends replies and exceptions to the output queue.

Create your local queues by using CSQUTIL utility or by using MQSeries operations and control panels from ISPF (csqorexx). The following is an example of the JCL that is used to create your local queues. In this example, MQND is the name of the queue manager:

```
/*
/* ADMIN_Q      : Admin queue
/* BACKOUT_Q    : Backout queue
/* IN_Q         : Input queue having a backout queue with threshold=3
/* REPLY_Q      : output queue or reply queue
/* DEADLETTER_Q: Dead letter queue
/*
/*DSNTECU EXEC PGM=CSQUTIL,PARM='MQND'
/*STEPLIB DD DSN=MQS.SCSQANLE,DISP=SHR
// DD DSN=MQS.SCSQAUTH,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
COMMAND DDNAME(CREATEQ)
/*
//CREATEQ DD *
```

```

DEFINE QLOCAL('ADMIN_Q') REPLACE +
DESCR('INPUT-OUTPUT') +
PUT(ENABLED) +
DEFPRTY(0) +
DEFPSIST(NO) +
SHARE +
DEFSOPT(SHARED) +
GET(ENABLED)
DEFINE QLOCAL('BACKOUT_Q') REPLACE +
DESCR('INPUT-OUTPUT') +
PUT(ENABLED) +
DEFPRTY(0) +
DEFPSIST(NO) +
SHARE +
DEFSOPT(SHARED) +
GET(ENABLED)
DEFINE QLOCAL('REPLY_Q') REPLACE +
DESCR('INPUT-OUTPUT') +
PUT(ENABLED) +
DEFPRTY(0) +
DEFPSIST(NO) +
SHARE +
DEFSOPT(SHARED) +
GET(ENABLED)
DEFINE QLOCAL('IN_Q') REPLACE +
DESCR('INPUT-OUTPUT') +
PUT(ENABLED) +
DEFPRTY(0) +
DEFPSIST(NO) +
SHARE +
DEFSOPT(SHARED) +
GET(ENABLED) +
BOQNAME('BACKOUT_Q') +
BOTHRESH(3)
DEFINE QLOCAL('DEADLETTER_Q') REPLACE +
DESCR('INPUT-OUTPUT') +
PUT(ENABLED) +
DEFPRTY(0) +
DEFPSIST(NO) +
SHARE +
DEFSOPT(SHARED) +
GET(ENABLED)
ALTER QMGR DEADQ ('DEADLETTER_Q') REPLACE
/*

```

Environment variables for logging and tracing MQListener:

Two environment variables control logging and tracing for MQListener. These variables are defined in the file `.profile`.

MQLSNTRC

When this ENV variable is set to 1, it will write function entry, data, and exit points to a unique HFS or zFS file. A unique trace file will be generated whenever any of the MQListener commands are run. This trace file will be used by IBM software support for debugging if the customer reports any problem. Unless requested, this variable should not be defined.

MQLSNLOG

The log file contains diagnostic information about the major events. This ENV variable is set to the name of the file where all log information will be written. All instances of MQListener daemon running one or more tasks will share the same file. For monitoring MQListener daemon, this variable should always be set. When MQListener daemon is running, open the

log/trace files only in read mode (use cat/more/tail commands in z/OS UNIX System Services to open the files) as they are used by the daemon process for writing.

Refer to the README file for more details about these variables.

Configuration table: SYSMQL.LISTENERS:

If you use MQListener, you must create the MQListener configuration table SYSMQL.LISTENERS by running installation job DSNTIJML.

The following table describes each of the columns of the configuration table SYSMQL.LISTENERS.

Table 142. Description of columns in the SYSMQL.LISTENERS table

Column name	Description
CONFIGURATIONNAME	The configuration name. The configuration name enables you to group several tasks into the same configuration. A single instance of MQListener can run all of the tasks that are defined within a configuration name.
QUEUEMANAGER	The name of the WebSphere MQ subsystem that contains the queues that are to be used.
INPUTQUEUE	The name of the queue in the WebSphere MQ subsystem that is to be monitored for incoming messages. The combination of the input queue and the queue manager are unique within a configuration
PROCNODE	Currently unused
PROCSHEMA	The schema name of the stored procedure that will be called by MQListener
PROCNAME	The name of the stored procedure that will be called by MQListener
PROCTYPE	Currently unused
NUMINSTANCES	The number of duplicate instances of a single task that are to run in this configuration
WAITMILLIS	The time MQListener waits (in milliseconds) after processing the current message before it looks for the next message
MINQUEDEPTH	Currently unused

Configuring MQListener tasks:

As part of configuring MQListener in DB2 for z/OS, you must configure at least one MQListener task.

Use the MQListener command, db2mqln1 or db2mqln2, to configure MQListener tasks. Issue the command from z/OS UNIX System Services command line in any directory. Alternatively, You can put the command in a file, grant execute permission and use the BPXBATCH utility to invoke the script from JCL. The sample script files are provided and are located in /MQListener-install-path/mqlsn/listener/script directory in z/OS UNIX System Services. Sample JCL (DSNTEJML) is also provided that invokes the script files and is located in prefix.SDSNSAMP. The add parameter with the db2mqln1 or db2mqln2 command updates a row in the DB2 table SYSMQL.LISTENERS.

- To add an MQListener configuration, issue the following command:

```
db2mq1n1/db2mq1n2 add
  -ssID <subsystem name>
  -config <configuration name>
  -queueManager <queuemanager name>
  -inputQueue <inputqueue name>
  -procName <stored-procedure name>
  -procSchema <stored-procedure schema name>
  -numInstances <number of instances>
```
- To display information about the configuration, issue the following command:

```
db2mq1n1/db2mq1n2 show
  -ssID <subsystem name>
  -config <configuration name>
```

To display information about all the configurations, issue the following command:

```
db2mq1n1/db2mq1n2 show
  -ssID <subsystem name>
  -config all
```

- To remove the messaging tasks, issue the following command:

```
db2mq1n1/db2mq1n2 remove
  -ssID <subsystem name>
  -config <configuration name>
  -queueManager <queuemanager name>
  -inputQueue <inputqueue name>
```
- To run the MQListener task, issue the following command:

```
db2mq1n1/db2mq1n2 run
  -ssID <subsystem name>
  -config <configuration name>
  -adminQueue <adminqueue name>
  -adminQMGr <adminqueuemanager name>
```
- To shutdown the MQListener daemon, issue the following command:

```
db2mq1n1/db2mq1n2 admin
  -adminQueue <adminqueue name>
  -adminQMGr <adminqueuemanager name>
  -adminCommand shutdown
```
- To restart the MQListener daemon, issue the following command:

```
db2mq1n1/db2mq1n2 admin
  -adminQueue <adminqueue name>
  -adminQMGr <adminqueuemanager name>
  -adminCommand restart
```
- To get help with the command and the valid parameters, issue the following command:

```
db2mq1n1/db2mq1n2 help
```
- To get help for a particular parameter, issue the following command, where 'command' is a specific parameter:

```
db2mq1n1/db2mq1n2 help <command>
```

Restriction:

- Use the same queue manager for the request queue and the reply queue.
- MQListener does not support logical messages that are composed of multiple physical messages. MQListener processes physical messages independently.

Creating a sample stored procedure to use with MQ Listener:

You can create a sample stored procedure, APROC, that can be used by MQListener to store a message in a table. The stored procedure returns the string OK if the message is successfully inserted into the table.

The following steps create DB2 objects that you can use with MQListener applications:

1. Create a table using SPUFI, DSNTEP2, or the command line processor in the subsystem where you want to run MQListener:

```
CREATE TABLE PROCTABLE (MSG VARCHAR(25) CHECK (MSG NOT LIKE 'FAIL%'));
```

The table contains a check constraint so that messages that start with the characters FAIL cannot be inserted into the table. The check constraint is used to demonstrate the behavior of MQListener when the stored procedure fails.

2. Create the following SQL procedure and define it to the same DB2 subsystem:

```
CREATE PROCEDURE TEST.APROC (  
    IN PIN VARCHAR(25),  
    OUT POUT VARCHAR(2))  
LANGUAGE SQL  
FENCED  
NOT DETERMINISTIC  
NO DBINFO  
COLLID TESTLSRN  
WLM ENVIRONMENT TESTWLMX  
ASUTIME NO LIMIT  
STAY RESIDENT NO  
PROGRAM TYPE MAIN  
SECURITY USER  
PROCEDURE1: BEGIN  
    INSERT INTO PROCTABLE VALUES(PIN);  
    SET POUT = 'OK';  
END PROCEDURE1
```

TESTLSRN is the name of the collection that is used for this stored procedure and TESTWLMX is the name of the WLM environment where this stored procedure will run.

3. Optional: Bind the collection TESTLSRN to the plan DB2MQLSN, which is used by MQListener:

```
BIND PLAN(DB2MQLSN)          +  
    PKLIST(LSNR.*,TESTLSRN.*) +  
    ACTION(REP) DISCONNECT(EXPLICIT);
```

If your application calls a stored procedure or user defined function that is defined with the COLLID option, the application does not need to include the collection ID in its plan. Thus, this step is optional.

MQListener error processing:

MQListener reads from WebSphere MQ message queues and calls DB2 stored procedures with those messages. If any errors occur during this process and the message is to be sent to the deadletter queue, MQListener returns a reason code to the deadletter queue.

Specifically, MQListener performs the following actions:

- prefixes the message with an MQ dead letter header (MQDLH) structure
- sets the reason field in the MQDLH structure to the appropriate reason code
- sends the message to the deadletter queue

The following table describes the reason codes that the MQListener daemon returns.

Table 143. Reason codes that MQListener returns

Reason code	Explanation
900	The call to a stored procedure was successful but an error occurred during the DB2 commit process and either of the following conditions were true: <ul style="list-style-type: none"> • No exception report was requested.¹ • An exception report was requested, but could not be delivered. This reason code applies only to one-phase commit environments.
901	The call to the specified stored procedure failed and the disposition of the MQ message is that an exception report be generated and the original message be sent the deadletter queue.
902	All of the following conditions occurred: <ul style="list-style-type: none"> • The disposition of the MQ message is that an exception report is not to be generated. ¹ • The stored procedure was called unsuccessfully the number of times that is specified as the backout threshold. • The name of the backout queue is the same as the deadletter queue. This reason code applies only to two-phase commit environments.
MQRC_TRUNCATED_MSG_FAILED	The size of the MQ message is greater than the input parameter of the stored procedure that is to be invoked. In one-phase commit environments, this oversized message is sent to the dead letter queue. In two-phase commit environments, this oversized message is sent to the deadletter queue only when the message cannot be delivered to the backout queue.

Note:

1. To specify that the receiver application generate exception reports if errors occur, set the report field in the MQMD structure that was used when sending the message to one of the following values:
 - MQRO_EXCEPTION
 - MQRO_EXCEPTION_WITH_DATA
 - MQRO_EXCEPTION_WITH_FULL_DATA

Related reference:

 [WebSphere MQ information center](#)

MQListener examples:

The application receives a message, inserts the message into a table, and generates a simple response message.

To simulate a processing failure, the application includes a check constraint on the table that contains the message. The constraint prevents any string that begins with the characters 'fail' from being inserted into the table. If you attempt to insert a message that violates the check constraint, the example application returns an error message and re-queues the failing message to the backout queue.

In this example, the following assumptions are made:

- MQListener is installed and configured for subsystem DB7A.
- MQND is the name of MQSeries subsystem that is defined. The Queue Manager is running, and the following local queues are defined in the DB7A subsystem:
 - ADMIN_Q : Admin queue

BACKOUT_Q : Backout queue
IN_Q : Input queue that has a backout queue withthreshold = 3
REPLY_Q : Output queue or Reply queue
DEADLLETTER_Q : Dead letter queue

- The person who is running the MQListener daemon has execute permission on the DB2MQLSN plan.

Before you run the MQListener daemon, add the following configuration, named ACFG, to the configuration table by issuing the following command:

```
db2mq1n2 add
  -ssID DB7A
  -config ACFG
  -queueManager MQND
  -inputQueue IN_Q
  -procName APROC
  -procSchema TEST
```

Run the MQListener daemon for two-phase commit for the added configuration 'ACFG'. To run MQListener with all of the tasks specified in a configuration, issue the following command:

```
db2mq1n2 run
  -ssID DB7A
  -config ACFG
  -adminQueue ADMIN_Q
  -adminQMgr MQND
```

The following examples show how to use MQListener to send a simple message and then inspect the results of the message in the WebSphere MQ queue manager and the database. The examples include queries to determine if the input queue contains a message or to determine if a record is placed in the table by the stored procedure.

MQListener example 1: Running a simple application:

1. Start with a clean database table by issuing the following SQL statement:

```
delete from PROCTABLE
```
2. Send a datagram to the input queue, 'IN_Q', with the message as 'sample message'. Refer to WebSphere MQ sample CSQ4BCK1 to send a message to the queue. Specify the MsgType option for 'Message Descriptor' as 'MQMT_DATAGRAM'.
3. Query the table by using the following statement to verify that the sample message is inserted:

```
select * from PROCTABLE
```
4. Display the number of messages that remain on the input queue to verify that the message has been removed. Issue the following command from a z/OS console:

```
/-MQND display queue('In_Q') curdepth
```

MQListener example 2: Sending requests to the input queue and inspecting the reply:

1. Start with a clean database table by issuing the following SQL statement:

```
delete from PROCTABLE
```
2. Send a request to the input queue, 'IN_Q', with the message as 'another sample message'. Refer to WebSphere MQ sample CSQ4BCK1 to send a message to the queue. Specify the MsgType option for 'Message Descriptor' as 'MQMT_REQUEST' and the queue name for ReplytoQ option.

3. Query the table by using the following statement to verify that the sample message is inserted:

```
select * from PROCTABLE
```
4. Display the number of messages that remain on the input queue to verify that the message has been removed. Issue the following command from a z/OS console:

```
/-MQND display queue('In_Q') curdepth
```
5. Look at the ReplytoQ name that you specified when you sent the request message for the reply by using the WebSphere MQ sample program CSQ4BCJ1. Verify that the string 'OK' is generated by the stored procedure.

MQListener example 3: Testing an unsuccessful insert operation: If you send a message that starts with the string 'fail', the constraint in the table definition is violated, and the stored procedure fails.

1. Start with a clean database table by issuing the following SQL statement:

```
delete from PROCTABLE
```
2. Send a request to the input queue, 'IN_Q', with the message as 'failing sample message'. Refer to WebSphere MQ sample CSQ4BCK1 to send a message to the queue. Specify the MsgType option for 'Message Descriptor' as 'MQMT_REQUEST' and the queue name for ReplytoQ option.
3. Query the table by using the following statement to verify that the sample message is not inserted:

```
select * from PROCTABLE
```
4. Display the number of messages that remain on the input queue to verify that the message has been removed. Issue the following command from a z/OS console:

```
/-MQND display queue('In_Q') curdepth
```
5. Look at the Backout queue and find the original message by using the WebSphere MQ sample program CSQ4BCJ1.

Note: In this example, if a request message with added options for 'exception report' is sent (the Report option is specified for 'Message Descriptor'), an exception report is sent to the reply queue and the original message is sent to the deadletter queue.

Chapter 16. DB2 as a web services consumer and provider

Web services are a set of resources and components that applications can use over HTTP. You can use DB2 as a web services provider and a web services consumer.

DB2 as a web services consumer

DB2 can act as a client for web services, which enables you to be a consumer of web services in your DB2 applications.

SOAP web services Simple Object Access Protocol (SOAP) is an XML protocol that consists of the following characteristics:

- An envelope that defines a framework for describing the contents of a message and how to process the message
- A set of encoding rules for expressing instances of application-defined data types
- A convention for representing SOAP requests and responses

A set of SOAP functions is provided by DB2 and is installed and configured when you install or migrate DB2.

DB2 as a web services provider

You can enable your DB2 data and applications as web services through the Web Services Object Runtime Framework (WORF). You can define a web service in DB2 by using a Document Access Definition Extension (DADX). In the DADX file, you can define web services based on SQL statements and stored procedures. Based on your definitions in the DADX file, WORF performs the following actions:

- Handles the connection to DB2 and the execution of the SQL and the stored procedure call
- Converts the result to a web service
- Handles the generation of any Web Services Definition Language (WSDL) and UDDI (Universal Description, Discovery, and Integration) information that the client application needs

For more information about using DB2 as a web services provider, see DB2 Information Integrator Application Developer's Guide.

Deprecated: The SOAPHTTPV and SOAPHTTPC user-defined functions

DB2 provides user-defined functions that allow you to work with SOAP and consume web services in SQL statements. The user-defined functions are two varieties of SOAPHTTPV for VARCHAR data and two varieties of SOAPHTTPC for CLOB data.

Restriction: SOAPHTTPV and SOAPHTTPC user-defined functions have been deprecated. Use SOAPHTTPNV and SOAPHTTPNC user-defined functions instead.

The user-defined functions perform the following actions:

1. Compose a SOAP request
2. Post the request to the service endpoint

3. Receive the SOAP response
4. Return the content of the SOAP body

When a consumer receives the result of a web services request, the SOAP envelope is stripped and the XML document is returned. An application program can process the result data and perform a variety of operations, including inserting or updating a table with the result data.

SOAPHTTPV and SOAPHTTPC are user-defined functions that enable DB2 to work with SOAP and to consume web services in SQL statements. These functions are overloaded functions that are used for VARCHAR or CLOB data of different sizes, depending on the SOAP body. Web services can be invoked in one of four ways, depending on the size of the input data and the result data. SOAPHTTPV returns VARCHAR(32672) data and SOAPHTTPC returns CLOB(1M) data. Both functions accept either VARCHAR(32672) or CLOB(1M) as the input body.

Example: The following example shows an HTTP post header that posts a SOAP request envelope to a host. The SOAP envelope body shows a temperature request for Barcelona.

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: services.xmethods.net
Connection: Keep-Alive User-Agent: DB2SOAP/1.0
Content-Type: text/xml; charset="UTF-8"
SOAPAction: ""
Content-Length: 410

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema >
  <SOAP-ENV:Body>
    <ns:getTemp xmlns:ns="urn:xmethods-Temperature">
      <city>Barcelona</city>
    </ns:getTemp>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example: The following example is the result of the preceding example. This example shows the HTTP response header with the SOAP response envelope. The result shows that the temperature is 85 degrees Fahrenheit in Barcelona.

```
HTTP/1.1 200 OK
Date: Wed, 31 Jul 2002 22:06:41 GMT
Server: Enhydra-MultiServer/3.5.2
Status: 200
Content-Type: text/xml; charset=utf-8
Servlet-Engine: Lutris Enhydra Application Server/3.5.2
  (JSP 1.1; Servlet 2.2; Java™ 1.3.1_04;
  Linux 2.4.7-10smp i386; java.vendor=Sun Microsystems Inc.)
Content-Length: 467
Set-Cookie: JSESSIONID=JLEcR34rBc2GTIkN-0F51ZDK;Path=/soap
X-Cache: MISS from www.xmethods.net
Keep-Alive: timeout=15, max=10
Connection: Keep-Alive

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema >
  <SOAP-ENV:Body>
    <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
```



```

        SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/ >
        <return xsi:type="xsd:float">85</return>
    </ns1:getTempResponse>
</SOAP-ENV:Body></SOAP-ENV:Envelope>

```

Example: The following example shows how to insert the result from a web service into a table

```

INSERT INTO MYTABLE(XMLCOL) VALUES (DB2XML.SOAPHTTPC(
    'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
    'http://tempuri.org/db2sample/list.dadx',
    '<listDepartments xmlns="http://tempuri.org/db2sample/listdadx">
        <deptno>A00</deptno>
    </ListDepartments>'))

```

The SOAPHTTPNV and SOAPHTTPNC user-defined functions

DB2 provides SOAPHTTPNV and SOAPHTTPNC user-defined functions that allow you to work with SOAP and consume web services in SQL statements. The user-defined functions are two varieties of SOAPHTTPNV for VARCHAR data and two varieties of SOAPHTTPNC for CLOB data.

The user-defined functions perform the following actions:

1. Post the input SOAP request to the service endpoint
2. Receive and return the SOAP response

SOAPHTTPNV and SOAPHTTPNC allow you to specify a complete SOAP message as input and return complete SOAP messages from the specified web service as a CLOB or VARCHAR representation of the returned XML data. SOAPHTTPNV returns VARCHAR(32672) data and SOAPHTTPNC returns CLOB(1M) data. Both functions accept either VARCHAR(32672) or CLOB(1M) as the input body.

SOAPHTTPNV and SOAPHTTPNC user-defined functions can support SOAP 1.1 or SOAP 1.2. Check with your system administrator to determine which levels of SOAP are supported by the user-defined functions in your environment.

Example


The following example shows how to insert the complete result from a web service into a table using SOAPHTTPNC.

```

INSERT INTO EMPLOYEE(XMLCOL)
VALUES (DB2XML.SOAPHTTPNC(
    'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
    'http://tempuri.org/db2sample/list.dadx',
    '<?xml version="1.0" encoding="UTF-8" ?>' ||
    '<SOAP-ENV:Envelope ' ||
    'xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ' ||
    'xmlns:xsd="http://www.w3.org/2001/XMLSchema" ' ||
    'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' ||
    '<SOAP-ENV:Body' ||
    '<listDepartments xmlns="http://tempuri.org/db2sample/list.dadx">
        <deptNo>A00</deptNo>
    </listDepartments>' ||
    '</SOAP-ENV:Body>' ||
    '</SOAP-ENV:Envelope>'))

```

Related tasks:

 Additional steps for enabling web service user-defined functions (DB2 Installation and Migration)

SQLSTATEs for DB2 as a web services consumer

DB2 returns SQLSTATE values for error conditions that are related to using DB2 as a web services consumer.

The following tables show possible SQLSTATE values.


Table 144. SQLSTATE values for SOAPHTTPV and SOAPHTTPC user-defined functions

SQLSTATE	Description
38301	An unexpected NULL value was pass as input to the function.
38302	The function was unable to allocate space.
38304	An unknown protocol was specified ion the endpoint URL.
38305	An invalid URL was specified on the endpoint URL.
38306	An error occurred while attempting to create a TCP/IP socket.
38307	An error occurred while attempting to bind a TCP/IP socket.
38308	The function could not resolve the specified host name.
38309	An error occurred while attempting to connect to the specified server.
38310	An error occurred while attempting to retrieve information from the protocol.
38311	An error occurred while attempting to set socket options.
38312	The function received unexpected data returned for the web service.
38313	The web service did not return data of the proper content type.
38314	An error occurred while initializing the XML parser.
38315	An error occurred while creating the XML parser.
38316	An error occurred while establishing a handler for the XML parser.
38317	The XML parser encountered an error while parsing the result data.
38318	The XML parser could not convert the result data to the database code page.
38319	The function could not allocate memory when creating a TCP/IP socket.
38320	An error occurred while attempting to send the request to the specified server.
38321	The function was unable to send the entire request to the specified server.
38322	An error occurred while attempting to read the result data from the specified server.
38323	An error occurred while waiting for data to be returned from the specified server.
38324	The function encountered an internal error while attempting to format the input message.
38325	The function encountered an internal error while attempting to add namespace information to the input message.
38327	The XML parser could not strip the SOAP envelope from the result message.
38328	An error occurred while processing an SSL connection.

Table 145. SQLSTATE values for SOAPHTTPNV and SOAPHTTPNC user-defined functions

SQLSTATE	Description
38350	An unexpected NULL value was specified for the endpoint, action, or SOAP input.
38351	A dynamic memory allocation error.
38352	An unknown or unsupported transport protocol.
38353	An invalid URL was specified.
38354	An error occurred while resolving the hostname.
38355	A memory exception for socket.
38356	An error occurred during socket connect.
38357	An error occurred while setting socket options.
38358	An error occurred during input/output control (ioctl) to verify HTTPS enablement.
38359	An error occurred while reading from the socket.
38360	An error occurred due to socket timeout.
38361	No response from the specified host.
38362	An error occurred due to an unexpected HTTP return or content type
38363	The TCP/IP stack was not enabled for HTTPS.

Related tasks:

 Additional steps for enabling web service user-defined functions (DB2 Installation and Migration)

Chapter 17. Preparing an application to run on DB2 for z/OS

To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

You can perform these steps by using one of the following methods:

Productivity hint: To avoid rework, first test your SQL statements using SPUFI. Then compile your program **without** SQL statements, and resolve all compiler errors. Finally, proceed with the preparation and the DB2 precompiler or with the host compiler that supports that DB2 coprocessor.

The following types of applications require different methods of program preparation:

- Applications that contain ODBC calls
- Applications in interpreted languages, such as REXX
- Java applications, which can contain JDBC calls or embedded SQL statements

For information about running REXX programs, which you do not prepare for execution, see “Running a DB2 REXX application” on page 1020.

Steps in program preparation:

The following topics provide details on preparing and running a DB2 application:

- “Processing SQL statements” on page 918
- “Compiling and link-editing an application” on page 942
- “Binding an application” on page 943
- Chapter 18, “Running an application on DB2 for z/OS,” on page 1017.

Binding a package is not necessary in all cases. These instructions assume that you bind some of your DBRMs into packages and include a package list in your plan.

If you use CICS, you might need additional steps; see:

- “Translating command-level statements in a CICS program” on page 929
- “Example of calling applications in a command procedure” on page 1030

For more information about when to bind a package, see “DB2 program preparation overview” on page 973.

Preparing applications by using JCL procedures:

A number of methods are available for preparing an application to run. You can:

- Use DB2 interactive (DB2I) panels, which lead you step by step through the preparation process.
- Submit a background job using JCL (which the program preparation panels can create for you).
- Start the DSNH CLIST in TSO foreground or background.
- Use TSO prompters and the DSN command processor.
- Use JCL procedures added to your SYS1.PROCLIB (or equivalent) at DB2 installation time.

- For C and C++ only, you can invoke the coprocessor from UNIX System Services and, if the DBRM is generated in a HFS file, you can use the command line processor to bind the resulting DBRM. Optionally, you can also copy the DBRM into a partitioned data set member by using the oput and oget commands and then bind it by using conventional JCL.

This topic describes how to use JCL procedures to prepare a program.

For information about using the DB2I panels, see Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915.

Preparing applications by the DB2 Program:

If you develop programs using TSO and ISPF, you can prepare them to run by using the DB2 Program Preparation panels. These panels guide you step by step through the process of preparing your application to run. Other ways of preparing a program to run are available, but using DB2 Interactive (DB2I) is the easiest because it leads you automatically from task to task.

Important: If your C++ program satisfies both of the following conditions, you must use a JCL procedure to prepare it:

- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

To prepare an application by using the DB2 Program Preparation panels:

1. If you want to display or suppress message IDs during program preparation, specify one of the following commands on the ISPF command line:

TSO PROFILE MSGID

Message IDs are displayed

TSO PROFILE NOMSGID

Message IDs are suppressed

2. Open the DB2I Primary Option Menu.
3. Select the option that corresponds to the Program Preparation panel.
4. Complete the Program Preparation panel and any subsequent panels. After you complete each panel, DB2I automatically displays the next appropriate panel.


Preparation guidelines for DL/I batch programs:

Use the following guidelines when you prepare a program to access DB2 and DL/I in a batch program:

- “Processing SQL statements by using the DB2 precompiler” on page 920
- “Binding a batch program” on page 956
- “Compiling and link-editing an application” on page 942
- “Loading and running a batch program” on page 1024

Related concepts:

 Command line processor (DB2 Commands)

 TSO attachment facility (Introduction to DB2 for z/OS)

Related reference:

 DSNH (TSO CLIST) (DB2 Commands)

Setting the DB2I defaults

When you use the DB2 Interactive (DB2I) panels to prepare an application, you can specify the default values that DB2I is to use. These default values can include the default application language and default JCL JOB statement. Otherwise, DB2I uses the system default values that were set at installation time.

To set the DB2I defaults:

As DB2I leads you through a series of panels, enter the default values that you want on the following panels when they are displayed.

Table 146. DB2I panels to use to set default values

If you want to set the following default values...	Use this panel
<ul style="list-style-type: none">• subsystem ID• number of additional times to attempt to connect to DB2• programming language• number of lines on each page of listing or SPUI output• lowest level of message to return to you during the BIND phase• SQL string delimiter for COBOL programs• how to represent decimal separators• smallest value of the return code (from precompile, compile, link-edit, or bind) that prevents later steps from running• default number of input entry rows to generate on the initial display of ISPF panels• user ID to associate with the trusted connection for the current DB2I session	DB2I Defaults Panel 1 panel
<ul style="list-style-type: none">• default JOB statement• symbol used to delimit a string in a COBOL statement in a COBOL application• whether DCLGEN generates a picture clause that has the form PIC G(n) DISPLAY-1 or PIC N(n).	DB2I Defaults Panel 2 panel

Table 146. DB2I panels to use to set default values (continued)

If you want to set the following default values...	Use this panel
The following package and plan characteristics	Defaults for Bind Package panel
<ul style="list-style-type: none"> • isolation level • whether to check authorization at run time or at bind time • when to release locks on resources • whether to obtain EXPLAIN information about how SQL statements in the plan or package execute • whether you need data currency for ambiguous cursors opened at remote locations • whether to use parallel processing • whether DB2 determines access paths at bind time and again at execution time • whether to defer preparation of dynamic SQL statements • whether DB2 keeps dynamic SQL statements after commit points • the application encoding scheme • whether you want to use optimization hints to determine access paths • when DB2 writes the changes for updated group buffer pool-dependent pages • whether run time (RUN) or bind time (BIND) rules apply to dynamic SQL statements at run time • whether to continue to create a package after finding SQL errors (packages only) • when to acquire locks on resources (plans only) • whether a CONNECT (Type 2) statement executes according to DB2 rules (DB2) or the SQL standard (STD). (plans only) • which remote connections end during a commit or a rollback (plans only) 	Defaults for Bind Plan panel

Related reference:

“DB2I Defaults Panel 1” on page 987

“DB2I Defaults Panel 2” on page 989

“Defaults for Bind Package and Defaults for Rebind Package panels” on page 998

“Defaults for Bind Plan and Defaults for Rebind Plan panels” on page 1001

Processing SQL statements

The first step in preparing an SQL application to run is to process the SQL statements in the program. To process the statements, use either the DB2 precompiler or the DB2 coprocessor. During this step, the SQL statements are replaced with calls to DB2 language interface modules, and a DBRM is created.

For assembler or Fortran applications, use the DB2 precompiler to prepare the SQL statements.

For C, C++, COBOL, or PL/I applications, you can use one of the following techniques to process SQL statements:

- Use the DB2 precompiler before you compile your program.
You can use this technique with any supported version of C or C++, COBOL, or PL/I.
- Invoke the DB2 coprocessor for the host language that you are using as you compile your program. You can use the DB2 coprocessor with C, C++, COBOL, and PL/I host compilers. To invoke the DB2 coprocessor, specify the SQL compiler option followed by its suboptions, which are those options that are defined for the DB2 precompiler. Some DB2 precompiler options are ignored.
 - For C or C++, you need IBM z/OS Version 1 Release 8 C/C++ or later. For C and C++, you can also invoke the coprocessor from UNIX System Services on z/OS to generate a DBRM in either a partitioned data set or an HFS file.
 - For COBOL, you need Enterprise COBOL for z/OS Version 3 Release 4 or later to use this technique.
 - For PL/I, you need Enterprise PL/I for z/OS Version 3 Release 4 or later to use this technique.

CICS: If the application contains CICS commands, you must translate the program before you compile it.

DB2 version in DSNHDECP module: When you process SQL statements in your program, if the DB2 version in DSNHDECP is the default system-provided version, DB2 issues a warning and processing continues. In this case, ensure that the information in DSNHDECP that DB2 uses accurately reflects your environment.

SQL statement processing:

Because most compilers do not recognize SQL statements, you can prevent compiler errors by using either the DB2 precompiler or the DB2 coprocessor.

The precompiler scans the program and returns modified source code, which you can then compile and link edit. The precompiler also produces a DBRM (database request module). You can bind this DBRM to a package using the BIND subcommand. When you complete these steps, you can run your DB2 application.

Alternatively, you can use the DB2 coprocessor for the host language. The DB2 coprocessor performs DB2 precompiler functions at compile time. When you use the DB2 coprocessor, the compiler (rather than the precompiler) scans the program and returns the modified source code. The DB2 coprocessor also produces a DBRM.

Related concepts:

 Using the DB2 C/C++ precompiler (XL C/C++ Programming Guide)

 DB2 coprocessor (Enterprise COBOL for z/OS Programming Guide)

“Differences between the DB2 precompiler and the DB2 coprocessor” on page 930

“DB2 program preparation overview” on page 973

Related tasks:

“Translating command-level statements in a CICS program” on page 929

Related reference:

 Enterprise COBOL for z/OS

Related information:

 DB2 Program Directory

Processing SQL statements by using the DB2 precompiler

The DB2 precompiler scans a program and copies all of the SQL statements and host variable information into a DBRM (database request module). The precompiler also returns source code that has been modified so that the SQL statements do not cause errors when you compile the program.

After the SQL statements and host variable information are copied into a DBRM and the modified source code is returned, you can compile and link-edit this modified source code.

Before you run the DB2 precompiler, use DCLGEN to obtain accurate SQL DECLARE TABLE statements. The precompiler checks table and column references against SQL DECLARE TABLE statements in the program, not the actual tables and columns.

DB2 does not need to be active when you precompile your program.

You do not need to precompile the program on the same DB2 subsystem on which you bind the DBRM and run the program. You can bind a DBRM and run it on a DB2 subsystem at the previous release level, if the original program does not use any properties of DB2 that are unique to the current release. You can also run applications on the current release that were previously bound on subsystems at the previous release level.

To process SQL statements by using the DB2 precompiler:

1. Ensure that your program is ready to be processed by the DB2 precompiler by performing the following actions: For information about the criteria for programs that are passed to the precompiler, see “Input to the DB2 precompiler” on page 923.
2. If you plan to run multiple precompilation jobs and are not using the DFSMSdfp partitioned data set extended (PDSE), change the DB2 language preparation procedures (DSNHCOB, DSNHCOB2, DSNHICOB, DSNHFOR, DSNHC, DSNHPLI, DSNHASM, DSNHSQL) to specify the DISP=OLD parameter instead of the DISP=SHR parameter. The DB2 language preparation procedures in job DSNTIJMV use the DISP=OLD parameter to enforce data integrity. However, the installation process converts the DISP=OLD parameter for the DBRM library data set to DISP=SHR, which can cause data integrity problems when you run multiple precompilation jobs.
3. Start the precompile process by using one of the following methods:
 - DB2I panels. Use the Precompile panel or the DB2 Program Preparation panels.
 - The DSNH command procedure (a TSO CLIST).
 - JCL procedures that are supplied with DB2. For more information about this method, see “DB2-supplied JCL procedures for preparing an application” on page 978.

Recommendation: Specify the SOURCE and XREF precompiler options to get complete diagnostic output from the DB2 precompiler. This output is useful if you need to precompile and compile program source statements several times before they are error-free and ready to link-edit.

The output that is returned from the DB2 precompiler is described in “Output from the DB2 precompiler” on page 925.

Preparing a program with object-oriented extensions by using JCL:

If your C++ or Enterprise COBOL for z/OS program satisfies both of these conditions, you need special JCL to prepare it:

- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

You must precompile the contents of each data set or member separately, but the prelinker must receive all of the compiler output together.

JCL procedure DSNHCPP2, which is in member DSNTIJMV of data set DSNA10.SDSNSAMP, shows you one way to do this for C++.

Precompiling a batch program: When you add SQL statements to an application program, you must precompile the application program and bind the resulting DBRM into a package, as described in Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915.

Related concepts:

“DCLGEN (declarations generator)” on page 161

Related reference:

 [DSNH \(TSO CLIST\) \(DB2 Commands\)](#)

Data sets that the precompiler uses

When you invoke the precompiler you need to provide data sets that contain input for the precompiler, such as the host programming statements and SQL statements. You also need to provide data sets where the precompiler can store its output, such as the modified source code and diagnostics messages.

Table 147. DD statements and data sets that the DB2 precompiler uses

DD statement	Data set description	Required?
DBRMLIB	Output data set, which contains the SQL statements and host variable information that the DB2 precompiler extracted from the source program. It is called Database Request Module (DBRM). This data set becomes the input to the DB2 bind process. The DCB attributes of the data set are RECFM FB, LRECL 80. DBRMLIB has to be a PDS and a member name must be specified. You can use IEBCOPY, IEHPROGM, TSO commands, COPY and DELETE, or PDS management tools for maintaining the data set.	Yes

Table 147. DD statements and data sets that the DB2 precompiler uses (continued)

DD statement	Data set description	Required?
STEPLIB	<p>Step library for the job step. In this DD statement, you can specify the name of the library for the precompiler load module, DSNHPC, and the name of the library for your DB2 application programming defaults member, DSNHDECP.</p> <p>Recommendation: Always use the STEPLIB DD statement to specify the library where your DB2 DSNHDECP module resides to ensure that the proper application defaults are used by the DB2 precompiler. The library that contains your DB2 DSNHDECP module needs to be allocated ahead of the prefix.SDSNLOAD library.</p>	No, but recommended
SYSCIN	<p>Output data set, which contains the modified source that the DB2 precompiler writes out. This data set becomes the input data set to the compiler or assembler. This data set must have attributes RECFM F or FB, and LRECL 80. SYSCIN can be a PDS or a sequential data set. If a PDS is used, the member name must be specified.</p>	Yes
SYSIN	<p>Input data set, which contains statements in the host programming language and embedded SQL statements. This data set must have the attributes RECFM F or FB, LRECL 80. SYSIN can be a PDS or a sequential data set. If a PDS is used, the member name must be specified.</p>	Yes

Table 147. DD statements and data sets that the DB2 precompiler uses (continued)

DD statement	Data set description	Required?
SYSLIB	INCLUDE library, which contains additional SQL and host language statements. The DB2 precompiler includes the member or members that are referenced by SQL INCLUDE statements in the SYSIN input from this DD statement. Multiple data sets can be specified, but they must be partitioned data sets with attributes RECFM F or FB, LRECL 80. SQL INCLUDE statements cannot be nested.	No
SYSPRINT	Output data set, which contains the output listing from the DB2 precompiler. This data set must have an LRECL of 133 and a RECFM of FBA. SYSPRINT must be a sequential data set	Yes
SYSTEMM	Terminal output file, which contains diagnostic messages from the DB2 precompiler. SYSTEMM) must be a sequential data set	No

Input to the DB2 precompiler

The primary input for the precompiler consists of statements in the host programming language and embedded SQL statements.

You can use the SQL INCLUDE statement to get secondary input from the include library, SYSLIB. The SQL INCLUDE statement reads input from the specified member of SYSLIB until it reaches the end of the member.

Another preprocessor, such as the PL/I macro preprocessor, can generate source statements for the precompiler. Any preprocessor that runs before the precompiler must be able to pass on SQL statements. Similarly, other preprocessors can process the source code, after you precompile and before you compile or assemble.

Input to the DB2 precompiler has the following restrictions:

- The size of a source program that DB2 can precompile is limited by the region size and the virtual memory available to the precompiler. These amounts vary with each system installation.
- The forms of source statements that can pass through the precompiler are limited. For example, constants, comments, and other source syntax that are not accepted by the host compilers (such as a missing right brace in C) can interfere with precompiler source scanning and cause errors. To check for such unacceptable source statements, run the host compiler before the precompiler. You can ignore the compiler error messages for SQL statements or comment out the SQL statements. After the source statements are free of unacceptable

compiler errors, you can then uncomment any SQL statements that you previously commented out and continue with the normal DB2 program preparation process for that host language.

- You must write host language statements and SQL statements using the same margins, as specified in the precompiler option MARGINS.
- The input data set, SYSIN, must have the attributes RECFM F or FB, LRECL 80.
- SYSLIB must be a partitioned data set, with attributes RECFM F or FB, LRECL 80.
- Input from the INCLUDE library cannot contain other precompiler INCLUDE statements.

Starting the precompiler dynamically when using JCL procedures

You can call the precompiler from an assembler program by using a macro.

You can call the precompiler from an assembler program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL.

To call the precompiler, specify DSNHPC as the entry point name. You can pass three address options to the precompiler; the following topics describe their formats. The options are addresses of:

- A precompiler option list
- A list of alternative DD names for the data sets that the precompiler uses
- A page number to use for the first page of the compiler listing on SYSPRINT

Related reference:

 [Using macros\(MVS Assembler Services Reference\)](#)

Precompiler option list format:

When you call the precompiler, you can specify a number of options, in a list, for SQL statement processing. You must specify that option list in a particular format.

The option list must begin on a 2-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list is EBCDIC and can contain precompiler option keywords, separated by one or more blanks, a comma, or both.

DD name list format:

When you call the precompiler, you can specify a list of alternative DD names for the data sets that the precompiler uses. You must specify this list in a particular format.

The DD name list must begin on a 2-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list is an 8-byte field, left-justified, and padded with blanks if needed.

The following table gives the following sequence of entries:

Table 148. DDNAME list entries

Entry	Standard ddname	Usage
1	Not applicable	
2	Not applicable	

Table 148. DDNAME list entries (continued)

Entry	Standard ddname	Usage
3	Not applicable	
4	SYSLIB	Library input
5	SYSIN	Source input
6	SYSPRINT	Diagnostic listing
7	Not applicable	
8	SYSUT1	Work data
9	SYSUT2	Work data
10	SYSUT3	Work data
11	Not applicable	
12	SYSTEM	Diagnostic listing
13	Not applicable	
14	SYSCIN	Changed source output
15	Not applicable	
16	DBRMLIB	DBRM output

Page number format:

When you call the precompiler, you can specify a page number to use for the first page of the compiler listing on SYSPRINT. You must specify this page number in a particular format.

A 6-byte field beginning on a 2-byte boundary contains the page number. The first 2 bytes must contain the binary value 4 (the length of the remainder of the field). The last 4 bytes contain the page number in character or zoned-decimal format.

The precompiler adds 1 to the last page number that is used in the precompiler listing and puts this value into the page-number field before returning control to the calling routine. Thus, if you call the precompiler again, page numbering is continuous.

Output from the DB2 precompiler

The major output from the DB2 precompiler is a database request module (DBRM). However, the DB2 precompiler also produces modified source statements, a list of source statements, a list of statements that refer to host names and columns, and diagnostics.

Specifically, the precompiler produces the following types of output:

listing output

The DB2 precompiler writes the following information in the SYSPRINT data set:

- Precompiler source listing

If the DB2 precompiler option SOURCE is specified, a source listing is produced. The source listing includes precompiler source statements, with line numbers that are assigned by the precompiler.

- Precompiler diagnostics

The precompiler produces diagnostic messages that include precompiler line numbers of statements that have errors.

- Precompiler cross-reference listing

If the DB2 precompiler option XREF is specified, a cross-reference listing is produced. The cross-reference listing shows the precompiler line numbers of SQL statements that refer to host names and columns.

The SYSPRINT data set has an LRECL of 133 and a RECFM of FBA. This data set uses the CCSID of the source program. Statement numbers in the output of the precompiler listing are displayed as they appear in the listing.

Terminal diagnostics

If a terminal output file, SYSTERM, exists, the DB2 precompiler writes diagnostic messages to it. A portion of the source statement accompanies the messages in this file. You can often use the SYSTERM file instead of the SYSPRINT file to find errors. This data set uses EBCDIC.

Modified source statements

The DB2 precompiler writes the source statements that it processes to SYSCIN, the input data set to the compiler or assembler. This data set must have attributes RECFM F or FB, and LRECL 80. The modified source code contains calls to the DB2 language interface. The SQL statements that the calls replace appear as comments. This data set uses the CCSID of the source program.

Database request modules

The database request module (DBRM) is a data set that contains the SQL statements and host variable information that is extracted from the source program, along with information that identifies the program and ties the DBRM to the translated source statements. It becomes the input to the bind process.

The data set requires space to hold all the SQL statements plus space for each host variable name and some header information. The header information alone requires approximately two records for each DBRM, 20 bytes for each SQL record, and 6 bytes for each host variable.

For an exact format of the DBRM, see the DBRM mapping macros, DSNXDBRM and DSNXNBRM, in library *prefix*.SDSNMACS. The DCB attributes of the data set are RECFM FB, LRECL 80. The precompiler sets the characteristics. You can use IEBCOPY, IEHPROGM, TSOCOPY and DELETE commands, or other PDS management tools for maintaining these data sets.

Restriction: Do not modify the contents of the DBRM. If you do, unpredictable results can occur. DB2 does not support modified DBRMs.

In a DBRM, the SQL statements and the list of host variable names use the UTF-8 character encoding scheme.

All other character fields in a DBRM use EBCDIC. The current release marker (DBRMMRIC) in the header of a DBRM is marked according to the release of the precompiler, regardless of the value of NEWFUN.

Processing SQL statements by using the DB2 coprocessor

As an alternative to the DB2 precompiler, you can use the DB2 coprocessor to process SQL statements. The DB2 coprocessor performs DB2 precompiler functions at compile time.

Exception: For PL/I, the DB2 coprocessor is called from the PL/I SQL preprocessor instead of the compiler.

The DB2 coprocessor has fewer restrictions on SQL programs than the DB2 precompiler. When you process SQL statements with the DB2 coprocessor, you can do the following things in your program:

- Use fully qualified names for structured host variables.
- Include SQL statements at any level of a nested program, instead of in only the top-level source file.(Although you can include SQL statements at any level of a nested program, you must compile the entire program as one unit.)
- Use nested SQL INCLUDE statements.
- **For C or C++ programs only:** Write applications with variable length format.
- **For C or C++ programs only:** Use codepage-dependent characters, such as left and right brackets, without using tri-graph notation when the programs use different code pages.

To process SQL statements by using the DB2 coprocessor, perform one of the following actions:

- Submit a JCL job to process that SQL statement. Include the following information:
 - Specify the SQL compiler option when you compile your program:
The SQL compiler option indicates that you want the compiler to invoke the DB2 coprocessor. Specify a list of SQL processing options in parentheses after the SQL keyword. Table 150 on page 933 lists the options that you can specify. For COBOL and PL/I, enclose the list of SQL processing options in single or double quotation marks. For PL/I, separate options in the list by a comma, blank, or both.

Examples:

C/C++

```
SQL(APOSTSQL STDSQL(NO))
```

COBOL

```
SQL("APOSTSQL STDSQL(NO)")
```

PL/I

```
PP(SQL("APOSTSQL,STDSQL(NO)"))
```

- For PL/I programs that use BIGINT or LOB data types, specify the following compiler options when you compile your program: LIMITS(FIXEDBIN(63), FIXEDDEC(31))
- If needed, increase the user's region size so that it can accommodate more memory for the DB2 coprocessor.
- Include DD statements for the following data sets in the JCL for your compile step:
 - DB2 load library (*prefix*.SDSNLOAD)
The DB2 coprocessor calls DB2 modules to process the SQL statements. You therefore need to include the name of the DB2 load library data set in the STEPLIB concatenation for the compiler step.
 - DBRM library
The DB2 coprocessor produces a DBRM. DBRMs and the DBRM library are described in “Output from the DB2 precompiler” on page 925. You need to include a DBRMLIB DD statement that specifies the DBRM library data set.
 - Library for SQL INCLUDE statements

If your program contains SQL INCLUDE *member-name* statements that specify secondary input to the source program, you need to also specify the data set for *member-name*. Include the name of the data set that contains *member-name* in the SYSLIB concatenation for the compiler step.

For C/C++ only: Invoke the DB2 coprocessor from UNIX System Services on z/OS. If you invoke the C/C++ DB2 coprocessor from UNIX System Services, you can choose to have the DBRM generated in a partitioned data set or an HFS file.

When you invoke the DB2 coprocessor, include the following information:

- Specify the SQL compiler option.
The SQL compiler option indicates that you want the compiler to invoke the DB2 coprocessor. Specify a list of SQL processing options in parentheses after the SQL keyword. Table 150 on page 933 lists the options that you can specify.
- Specify a location for the DBRM as the parameter for the dbrmlib option. You can specify one of the following items:
 - The name of a partitioned data set

Example: The following example invokes the C/C++ DB2 coprocessor to compile (with the c89 compiler) a sample C program and requests that the resulting DBRM is stored in the test member of the userid.dbrmlib.data data set:

```
c89 -Wc,"sql,dbrmlib(//'userid.dbrmlib.data(test)'),langlvl(extended)" -c t.c
```

- The name of an HFS file

The file name can be qualified, partially qualified, or unqualified. The file path can contain a maximum of 1024 characters, and the file name can contain a maximum of 255 characters. The first 8 characters of the file name, not including the file extension, should be unique within the file system.

Assume that your directory structure is /u/USR001/c/example and that your current working directory is /u/USR001/c. The following table shows examples of how to specify the HFS file names with the dbrmlib option and how the file names are resolved.

Table 149. How to specify HFS files to store DBRMs

If you specify...	The DBRM is generated in...
dbrmlib(/u/USR001/sample.dbrm)	/u/USR001/sample.dbrm
dbrmlib(example/sample.dbrm)	/u/USR001/c/example/sample.dbrm
dbrmlib(./sample.dbrm)	/u/USR001/sample.dbrm
dbrmlib(sample.dbrm)	/u/USR001/c/sample.dbrm

Example: The following example invokes the DB2 coprocessor to compile (with the c89 compiler) a sample C program and requests that the resulting DBRM is stored in the file test.dbrm in the tmp directory:

```
c89 -Wc,"sql,dbrmlib(/tmp/test.dbrm),langlvl(extended)" -c t.c
```

If you request that the DBRM be generated in an HFS file, you can bind the resulting DBRM by using the command line processor BIND command. For more information about using the command line processor BIND command, see "Binding a DBRM that is in an HFS file to a package or collection" on page 946

page 946. Optionally, you can also copy the DBRM into a partitioned data set member by using the `oput` and `oget` commands and then bind the DBRM by using conventional JCL.

Support for compiling a COBOL program that includes SQL from an assembler program

The COBOL compiler provides a facility that enables you to invoke the COBOL compiler by using an assembler program.

If you intend to use the DB2 coprocessor and start the COBOL compiler from an assembler program as part of your DB2 application preparation, you can use the SQL compiler option and provide the alternate DBRMLIB DD name the same way that you can specify other alternate DD names. The DB2 coprocessor creates the DBRM member according to your DBRM PDS library and the DBRM member that you specified using the alternate DBRMLIB DD name.

To use the alternate DBRMLIB DD name, Enterprise COBOL V4.1 and above is required.

Related reference:

 [IBM System z Enterprise Development Tools & Compilers](#)

Translating command-level statements in a CICS program

You can translate CICS applications with the CICS command language translator as a part of the program preparation process. CICS command language translators are available only for assembler, C, COBOL, and PL/I languages.

CICS:

Prepare your CICS program in either of these sequences:

Use the DB2 precompiler first, followed by the CICS Command Language Translator. This sequence is the preferred method of program preparation and the one that the DB2I Program Preparation panels support. If you use the DB2I panels for program preparation, you can specify translator options automatically, rather than needing to provide a separate option string.

Use the CICS command language translator first, followed by the DB2 precompiler. This sequence results in a warning message from the CICS translator for each EXEC SQL statement that it encounters. The warning messages have no effect on the result. If you are using double-byte character sets (DBCS), precompiling is recommended before translating, as described previously.

Program and process requirements: Use the DB2 precompiler before the CICS translator to prevent the precompiler from mistaking CICS translator output for graphic data.

If your source program is in COBOL, you must specify a string delimiter that is the same for the DB2 precompiler, COBOL compiler, and CICS translator. The defaults for the DB2 precompiler and COBOL compiler are not compatible with the default for the CICS translator.

If the SQL statements in your source program refer to host variables that a pointer stored in the CICS TWA addresses, you must make the host variables addressable

to the TWA before you execute those statements. For example, a COBOL application can issue the following statement to establish addressability to the TWA:

```
EXEC CICS ADDRESS
      TWA (address-of-twa-area)
END-EXEC
```

You can run CICS applications only from CICS address spaces. This restriction applies to the RUN option on the second program DSN command processor. All of those possibilities occur in TSO.

To prepare an application program, you can append JCL from a job that is created by the DB2 Program Preparation panels to the JCL for the CICS command language translator. To run the prepared program under CICS, you might need to define programs and transactions to CICS. Your system programmer must make the appropriate CICS resource or table entries.

prefix.SDSNSAMP contains examples of the JCL that is used to prepare and run a CICS program that includes SQL statements. The set of JCL includes:

- PL/I macro phase
- DB2 precompiling
- CICS Command Language Translation
- Compiling of the host language source statements
- Link-editing of the compiler output
- Binding of the DBRM
- Running of the prepared application.

Related reference:

“Sample applications in CICS” on page 1100

Related information:

 Resource Definition Guide (CICS Transaction Server for z/OS)

Differences between the DB2 precompiler and the DB2 coprocessor

The DB2 precompiler and DB2 coprocessor have architectural differences. You cannot switch from one to the other without considering those differences and adjusting your program accordingly.

Recommendation: Use the coprocessor instead of the precompiler when using Unicode variables in COBOL or PL/I applications.

Depending on whether you use the DB2 precompiler or the DB2 coprocessor, ensure that you account for the following differences:

- Differences in handling source CCSIDs:

The DB2 precompiler and DB2 coprocessor convert the SQL statements of your source program to UTF-8 for parsing.

The precompiler or DB2 coprocessor uses the source CCSID(*n*) value to convert from that CCSID to CCSID 1208 (UTF-8). The CCSID value must be an EBCDIC CCSID. If you want to prepare a source program that is written in a CCSID that cannot be directly converted to or from CCSID 1208, you must create an indirect conversion.

- Differences in handling host variable CCSIDs:
 - **COBOL:**

DB2 precompiler:

The DB2 precompiler sets CCSIDs for alphanumeric host variables only when the program includes an explicit DECLARE :hv VARIABLE statement.

DB2 coprocessor:

The COBOL compiler with National Character Support always sets CCSIDs for alphanumeric variables, including host variables that are used within SQL, to the source CCSID. Alternatively, you can specify that you want the COBOL DB2 coprocessor to handle CCSIDs the same way as the precompiler.

Recommendation: If you have problems with host variable CCSIDs, use the DB2 precompiler or change your application to include the DECLARE :hv VARIABLE statement to overwrite the CCSID that is specified by the COBOL compiler.

Example: Assume that DB2 has mapped a FOR BIT DATA column to a host variable in the following way:

```
01 hv1 pic x(5).
01 hv2 pic x(5).
```

```
EXEC SQL CREATE TABLE T1 (colwbit char(5) for bit data,
                          rowid char(5)) END-EXEC.
```

```
EXEC SQL
INSERT INTO T1 VALUES (:hv1, :hv2)
END-EXEC.
```

DB2 precompiler: In the modified source from the DB2 precompiler, hv1 and hv2 are represented to DB2 through SQLDA in the following way, without CCSIDs:

```
for hv1: NO CCSID
```

```
20 SQL-PVAR-NAME1 PIC S9(4) COMP-4 VALUE +0.
20 SQL-PVAR-NAMEC1 PIC X(30) VALUE ' '.
```

```
for hv2: NO CCSID
```

```
20 SQL-PVAR-NAME2 PIC S9(4) COMP-4 VALUE +0.
20 SQL-PVAR-NAMEC2 PIC X(30) VALUE ' '.
```

DB2 coprocessor: In the modified source from the DB2 coprocessor with the National Character Support for COBOL, hv1 and hv2 are represented to DB2 in the following way, with CCSIDs: (Assume that the source CCSID is 1140.) for hv1 and hv2, the value for CCSID is set to '1140' ('474'x) in input SQLDA of the INSERT statement.

```
'7F00000474000000007F'x
```

To ensure that no discrepancy exists between the column with FOR BIT DATA and the host variable with CCSID 1140, add the following statement for :hv1 or use the DB2 precompiler:

```
EXEC SQL DECLARE : hv1 VARIABLE FOR BIT DATA END-EXEC.
```

for hv1 declared with for bit data. The value in SQL---AVAR-NAME-DATA is set to 'FFFF'x for CCSID instead of '474x'.

```
'7F0000FFFF00000007F'x <<= with DECLARE :hv1 VARIABLE FOR BIT DATA  
vs.  
'7F00000474000000007F'x <<= without
```

– PL/I

DB2 coprocessor:

You can specify whether CCSIDs are to be associated with host variables by using the following PL/I SQL preprocessor options:

CCSID0

Specifies that the PL/I SQL preprocessor is not to set the CCSIDs for all host variables unless they are defined with the SQL DECLARE :hv VARIABLE statement.

NOCCSID0

Specifies that the PL/I SQL preprocessor is to set the CCSIDs for all host variables.

Related concepts:

[z/OS: Unicode Services User's Guide and Reference](#)

Related reference:

“Descriptions of SQL processing options”

[Enterprise COBOL for z/OS](#)

[SQL preprocessor options \(PL/I\) \(Enterprise PL/I for z/OS Programming Guide:\)](#)

Options for SQL statement processing

Use SQL processing options to specify how the DB2 precompiler and the DB2 coprocessor interpret and process input, and how they present output.

If you are using the DB2 precompiler, specify SQL processing options in one of the following ways:

- With DSNH operands
- With the PARM.PC option of the EXEC JCL statement
- On DB2I panels

If you are using the DB2 coprocessor, specify SQL processing options in one of the following ways:

- For C or C++, specify the options as the argument of the SQL compiler option.
- For COBOL, specify the options as the argument of the SQL compiler option.
- For PL/I, specify the options as the argument of the PP(SQL('option,...')) compiler option.

For examples of how to specify the DB2 coprocessor options, see “Processing SQL statements by using the DB2 coprocessor” on page 926

DB2 assigns default values for any SQL processing options for which you do not explicitly specify a value. Those defaults are the values that are specified on the APPLICATION PROGRAMMING DEFAULTS installation panels.

Descriptions of SQL processing options

You can specify any SQL processing options regardless of whether you use the DB2 precompiler or the DB2 coprocessor. However, the DB2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

The following table shows the options that you can specify when you use the DB2 precompiler or DB2 coprocessor. The table also includes abbreviations for those options and indicates which options are ignored for a particular host language or by the DB2 coprocessor. This table uses a vertical bar (|) to separate mutually exclusive options, and brackets ([]) to indicate that you can sometimes omit the enclosed option.

Table 150. SQL processing options

Option keyword	Meaning
APOST ¹	<p>Indicates that the DB2 precompiler is to use the apostrophe (') as the string delimiter in host language statements that it generates.</p> <p>This option is not available in all languages.</p> <p>APOST and QUOTE are mutually exclusive options. The default is in the field STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If STRING DELIMITER is the apostrophe ('), APOST is the default.</p>
APOSTSQL	<p>Recognizes the apostrophe (') as the string delimiter and the double quotation mark (") as the SQL escape character within SQL statements.</p> <p>APOSTSQL and QUOTESQL are mutually exclusive options. The default is in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If SQL STRING DELIMITER is the apostrophe ('), APOSTSQL is the default.</p>
ATTACH(TSO CAF RRSAF)	<p>Specifies the attachment facility that the application uses to access DB2. TSO, CAF, and RRSF applications that load the attachment facility can use this option to specify the correct attachment facility, instead of coding a dummy DSNHLI entry point.</p> <p>This option is not available for Fortran applications.</p> <p>The default is ATTACH(TSO).</p>

Table 150. SQL processing options (continued)

Option keyword	Meaning
CCSID(<i>n</i>) ⁵	<p data-bbox="511 254 1425 310">Specifies the numeric value <i>n</i> of the CCSID in which the source program is written. The number <i>n</i> must be an EBCDIC CCSID.</p> <p data-bbox="511 338 1425 394">The default setting is the EBCDIC system CCSID as specified on the panel DSNTIPF during installation.</p> <p data-bbox="511 422 1425 478">The DB2 coprocessor uses the following process to determine the CCSID of the source statements:</p> <ol data-bbox="511 485 1425 961" style="list-style-type: none"> 1. If the CCSID of the source program is specified by a compiler option, such as the COBOL CODEPAGE compiler option, the DB2 coprocessor uses that CCSID. 2. If the CCSID is not specified by a compiler option: <ol data-bbox="548 590 1425 961" style="list-style-type: none"> a. If the CCSID suboption of the SQL compiler option is specified and contains a valid EBCDIC CCSID, that CCSID is used. b. If the CCSID suboption of the SQL compiler option is not specified, and the compiler supports an option for specifying the CCSID, such as the COBOL CODEPAGE compiler option, the default for the CCSID compiler option is used. c. If the CCSID suboption of the SQL compiler option is not specified, and the compiler does not support an option for specifying the CCSID, the default CCSID from DSNHDECP or a user-specified application defaults module is used. d. If the CCSID suboption of the SQL option is specified and contains an invalid CCSID, compilation terminates.
COMMA	<p data-bbox="511 989 1425 1016">CCSID supersedes the GRAPHIC and NOGRAPHIC SQL processing options.</p> <p data-bbox="511 1043 1425 1100">If you specify CCSID(1026) or CCSID(1155), the DB2 coprocessor does not support the code point 'FC'X for the double quotation mark (").</p> <p data-bbox="511 1115 1425 1171">Recognizes the comma (,) as the decimal point indicator in decimal or floating point literals in the following cases:</p> <ul data-bbox="511 1178 1425 1297" style="list-style-type: none"> • For static SQL statements in COBOL programs • For dynamic SQL statements, when the value of installation parameter DYNRULS is NO and the package or plan that contains the SQL statements has DYNAMICRULES bind, define, or invoke behavior. <p data-bbox="511 1325 1425 1409">COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on Application Programming Defaults Panel 1 during installation.</p>
CONNECT(2 1) CT(2 1)	<p data-bbox="511 1430 1425 1457">Determines whether to apply type 1 or type 2 CONNECT statement rules.</p> <p data-bbox="548 1463 1425 1520">CONNECT(2) Default: Apply rules for the CONNECT (Type 2) statement CONNECT(1) Apply rules for the CONNECT (Type 1) statement</p> <p data-bbox="511 1535 1425 1587">If you do not specify the CONNECT option when you precompile a program, the rules of the CONNECT (Type 2) statement apply.</p>
DATE(ISO USA EUR JIS LOCAL)	<p data-bbox="511 1608 1425 1665">Specifies that date output should always be returned in a particular format, regardless of the format that is specified as the location default.</p> <p data-bbox="511 1692 1425 1749">The default is specified in the field DATE FORMAT on Application Programming Defaults Panel 2 during installation.</p> <p data-bbox="511 1776 1425 1854">The default format is determined by the installation defaults of the system where the program is bound, not by the installation defaults of the system where the program is precompiled.</p> <p data-bbox="511 1881 1425 1904">You cannot use the LOCAL option unless you have a date exit routine.</p>

Table 150. SQL processing options (continued)

Option keyword	Meaning
DEC(15 31) DEC15 DEC31 D15.s D31.s	<p>Specifies the maximum precision for decimal arithmetic operations.</p> <p>The default is in the field DECIMAL ARITHMETIC on Application Programming Defaults Panel 1 during installation.</p> <p>If the form <i>Dpp.s</i> is specified, <i>pp</i> must be either 15 or 31, and <i>s</i>, which represents the minimum scale to be used for division, must be a number between 1 and 9.</p>
DECP(<i>name</i>)	<p><i>name</i> represents the 1 to 8 character name of the application defaults data-only load module that is to be used.</p> <p>The default name DSNHDECP is used if this parameter is omitted.</p>
FLAG(I W E S) ¹	<p>Suppresses diagnostic messages below the specified severity level (Informational, Warning, Error, and Severe error for severity codes 0, 4, 8, and 12 respectively).</p> <p>The default setting is FLAG(I).</p>
FLOAT(S390 IEEE)	<p>Determines whether the contents of floating-point host variables in assembler, C, C++, or PL/I programs are in IEEE floating-point format or z/Architecture hexadecimal floating-point format. DB2 ignores this option if the value of HOST is anything other than ASM, C, CPP, or PLI.</p> <p>The default setting is FLOAT(S390).</p>
GRAPHIC	<p>This option is no longer used for SQL statement processing. Use the CCSID option instead.</p> <p>Indicates that the source code might use mixed data, and that X'0E' and X'0F' are special control characters (shift-out and shift-in) for EBCDIC data.</p> <p>GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is specified in the field MIXED DATA on Application Programming Defaults Panel 1 during installation.</p>

Table 150. SQL processing options (continued)

Option keyword	Meaning
HOST ¹ (ASM C[(FOLD)] CPP[(FOLD)] IBMCOB PLI FORTRAN SQL SQLPL)	<p>Defines the host language that contains the SQL statements.</p> <p>Use IBMCOB for Enterprise COBOL for z/OS.</p> <p>For C, specify:</p> <ul style="list-style-type: none"> • C if you do not want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase • C(FOLD) if you want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase <p>For C++, specify:</p> <ul style="list-style-type: none"> • CPP if you do not want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase • CPP(FOLD) if you want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase <p>For SQL procedural language, specify:</p> <ul style="list-style-type: none"> • SQL, to perform syntax checking and conversion to a generated C program for an external SQL procedure. • SQLPL, to perform syntax checking for a native SQL procedure. <p>If you omit the HOST option, the DB2 precompiler issues a level-4 diagnostic message and uses the default value for this option.</p> <p>The default is in the field LANGUAGE DEFAULT on Application Programming Defaults Panel 1 during installation.</p> <p>This option also sets the language-dependent defaults.</p>
LEVEL[(aaaa)] L	<p>Defines the level of a module, where <i>aaaa</i> is any alphanumeric value of up to seven characters. This option is not recommended for general use, and the DSNH CLIST and the DB2I panels do not support it.</p> <p>For assembler, C, C++, Fortran, and PL/I, you can omit the suboption (<i>aaaa</i>). The resulting consistency token is blank. For COBOL, you need to specify the suboption.</p>
LINECOUNT ¹ (<i>n</i>) LC	<p>Defines the number of lines per page to be <i>n</i> for the DB2 precompiler listing. This includes header lines that are inserted by the DB2 precompiler. The default setting is LINECOUNT(60).</p>
MARGINS ¹ (<i>m,n,c</i>) MAR	<p>Specifies what part of each source record contains host language or SQL statements. For assembler, this option also specifies where column continuations begin. The first option (<i>m</i>) is the beginning column for statements. The second option (<i>n</i>) is the ending column for statements. The third option (<i>c</i>) specifies where assembler continuations begin. Otherwise, the DB2 precompiler places a continuation indicator in the column immediately following the ending column. Margin values can range from 1 to 80.</p> <p>Default values depend on the HOST option that you specify.</p> <p>The DSNH CLIST and the DB2I panels do not support this option. In assembler, the margin option must agree with the ICTL instruction, if presented in the source.</p>

Table 150. SQL processing options (continued)

Option keyword	Meaning
NEWFUN(V8 V9 V10)	<p>Indicates whether to accept the function syntax that is new for the current version of DB2.</p> <p>NEWFUN(V8) Specifies that any syntax up to V8 will be allowed.</p> <p>NEWFUN(V9) Specifies that any syntax up to V9 will be allowed. NEWFUN(V9) is equivalent to NEWFUN(NO)</p> <p>NEWFUN(V10) Specifies that any syntax up to V10 will be allowed. NEWFUN(V10) is equivalent to NEWFUN(YES).</p> <p>The NEWFUN option applies only to the precompilation process by either the precompiler or the DB2 coprocessor, regardless of the current migration mode. You are responsible for ensuring that you bind the resulting DBRM on a subsystem in the correct migration mode.</p> <p>NEWFUN(YES) and NEWFUN(NO) options are deprecated.</p>
NOFOR	<p>In static SQL, eliminates the need for the FOR UPDATE or FOR UPDATE OF clause in DECLARE CURSOR statements. When you use NOFOR, your program can make positioned updates to any columns that the program has DB2 authority to update.</p> <p>When you do not use NOFOR, if you want to make positioned updates to any columns that the program has DB2 authority to update, you need to specify FOR UPDATE with no column list in your DECLARE CURSOR statements. The FOR UPDATE clause with no column list applies to static or dynamic SQL statements.</p> <p>Regardless of whether you use NOFOR, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns that are named in the clause, and you can specify the acquisition of update locks.</p> <p>You imply NOFOR when you use the option STDSQL(YES).</p> <p>If the resulting DBRM is very large, you might need extra storage when you specify NOFOR or use the FOR UPDATE clause with no column list.</p>
NOGRAPHIC	<p>This option is no longer used for SQL statement processing. Use the CCSID option instead.</p> <p>Indicates the use of X'0E' and X'0F' in a string, but not as control characters.</p> <p>GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is specified in the field MIXED DATA on Application Programming Defaults Panel 1 during installation.</p> <p>The NOGRAPHIC option applies to only EBCDIC data.</p>
NOOPTIONS NOOPTN	<p>Suppresses the DB2 precompiler options listing.</p>
NOPADNTSTR	<p>Indicates that output host variables that are NUL-terminated strings are not padded with blanks. That is, additional blanks are not inserted before the NUL-terminator is placed at the end of the string.</p> <p>PADNTSTR and NOPADNTSTR are mutually exclusive options. The default (PADNTSTR or NOPADNTSTR) is specified in the field PAD NUL-TERMINATED on Application Programming Defaults Panel 2 during installation.</p> <p>This option applies to only C and C++ applications.</p>

Table 150. SQL processing options (continued)

Option keyword	Meaning
NOSOURCE ² NOS	Suppresses the DB2 precompiler source listing. This is the default.
NOXREF	Suppresses the DB2 precompiler cross-reference listing. This is the default.
ONEPASS ON	Processes in one pass, to avoid the additional processing time for making two passes. Declarations must appear before SQL references. Default values depend on the HOST option specified. ONEPASS and TWOPASS are mutually exclusive options.
OPTIONS ¹ OPTN	Lists DB2 precompiler options. This is the default.
PADNTSTR	Indicates that output host variables that are NUL-terminated strings are padded with blanks with the NUL-terminator placed at the end of the string. PADNTSTR and NOPADNTSTR are mutually exclusive options. The default (PADNTSTR or NOPADNTSTR) is specified in the field PAD NUL-TERMINATED on Application Programming Defaults Panel 2 during installation. This option applies to only C and C++ applications.
PERIOD	Recognizes the period (.) as the decimal point indicator in decimal or floating point literals in the following cases: <ul style="list-style-type: none"> • For static SQL statements in COBOL programs • For dynamic SQL statements, when the value of installation parameter DYNRULS is NO and the package or plan that contains the SQL statements has DYNAMICRULES bind, define, or invoke behavior. <p>COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is specified in the field DECIMAL POINT IS on Application Programming Defaults Panel 1 during installation.</p>
QUOTE ¹ Q	Indicates that the DB2 precompiler is to use the quotation mark (") as the string delimiter in host language statements that it generates. QUOTE is valid only for COBOL applications. QUOTE is not valid for either of the following combinations of precompiler options: <ul style="list-style-type: none"> • CCSID(1026) and HOST(IBMCOB) • CCSID(1155) and HOST(IBMCOB) <p>The default is specified in the field STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If STRING DELIMITER is the double quotation mark (") or DEFAULT, QUOTE is the default. APOST and QUOTE are mutually exclusive options.</p>
QUOTESQL	Recognizes the double quotation mark (") as the string delimiter and the apostrophe (') as the SQL escape character within SQL statements. This option applies only to COBOL. The default is specified in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If SQL STRING DELIMITER is the double quotation mark (") or DEFAULT, QUOTESQL is the default. APOSTSQL and QUOTESQL are mutually exclusive options.
SOURCE ¹ S	Lists DB2 precompiler source and diagnostics.

Table 150. SQL processing options (continued)

Option keyword	Meaning
SQL(ALL DB2)	<p data-bbox="542 258 1430 312">Indicates whether the source contains SQL statements other than those recognized by DB2 for z/OS.</p> <p data-bbox="542 342 1455 596">SQL(ALL) is recommended for application programs whose SQL statements must execute on a server other than DB2 for z/OS using DRDA access. SQL(ALL) indicates that the SQL statements in the program are not necessarily for DB2 for z/OS. Accordingly, the SQL statement processor then accepts statements that do not conform to the DB2 syntax rules. The SQL statement processor interprets and processes SQL statements according to distributed relational database architecture (DRDA) rules. The SQL statement processor also issues an informational message if the program attempts to use IBM SQL reserved words as ordinary identifiers. SQL(ALL) does not affect the limits of the SQL statement processor.</p> <p data-bbox="542 625 1443 705">SQL(DB2), the default, means to interpret SQL statements and check syntax for use by DB2 for z/OS. SQL(DB2) is recommended when the database server is DB2 for z/OS.</p>
STDSQL(NO YES) ³	<p data-bbox="542 724 1227 747">Indicates to which rules the output statements should conform.</p> <p data-bbox="542 777 1458 856">STDSQL(YES)³ indicates that the precompiled SQL statements in the source program conform to certain rules of the SQL standard. STDSQL(NO) indicates conformance to DB2 rules.</p> <p data-bbox="542 886 1341 945">The default is specified in the field STD SQL LANGUAGE on Application Programming Defaults Panel 2 during installation.</p> <p data-bbox="542 974 1149 995">STDSQL(YES) automatically implies the NOFOR option.</p>
TIME(ISO USA EUR JIS LOCAL)	<p data-bbox="542 1014 1409 1066">Specifies that time output always return in a particular format, regardless of the format that is specified as the location default.</p> <p data-bbox="542 1096 1422 1148">The default is specified in the field TIME FORMAT on Application Programming Defaults Panel 2 during installation.</p> <p data-bbox="542 1178 1422 1260">The default format is determined by the installation defaults of the system where the program is bound, not by the installation defaults of the system where the program is precompiled.</p> <p data-bbox="542 1289 1308 1314">You cannot use the LOCAL option unless you have a time exit routine.</p>
TWOPASS TW	<p data-bbox="542 1333 1422 1386">Processes in two passes, so that declarations need not precede references. Default values depend on the HOST option that is specified.</p> <p data-bbox="542 1415 1166 1438">ONEPASS and TWOPASS are mutually exclusive options.</p> <p data-bbox="542 1467 1412 1549">For the DB2 coprocessor, you can specify the TWOPASS option for only PL/I applications. For C/C++ and COBOL applications, the DB2 coprocessor uses the ONEPASS option.</p>
VERSION(<i>aaaa</i> AUTO)	<p data-bbox="542 1568 1414 1621">Defines the version identifier of a package, program, and the resulting DBRM. A version identifier is an SQL identifier of up to 64 EBCDIC bytes.</p> <p data-bbox="542 1650 1425 1732">When you specify VERSION, the SQL statement processor creates a version identifier in the program and DBRM. This affects the size of the load module and DBRM. DB2 uses the version identifier when you bind the DBRM to a package.</p> <p data-bbox="542 1761 1450 1902">If you do not specify a version at precompile time, an empty string is the default version identifier. If you specify AUTO, the SQL statement processor uses the consistency token to generate the version identifier. If the consistency token is a timestamp, the timestamp is converted into ISO character format and is used as the version identifier. The timestamp that is used is based on the store clock value.</p>

Table 150. SQL processing options (continued)

Option keyword	Meaning
XREF ⁶	Includes a sorted cross-reference listing of symbols that are used in SQL statements in the listing output.

Notes:

1. The DB2 coprocessor ignores this option when the DB2 coprocessor is invoked by the compiler to prepare the application.
2. This option is always in effect when the DB2 coprocessor is invoked by the compiler to prepare the application.
3. You can use STDSQL(86) as in prior releases of DB2. The SQL statement processor treats it the same as STDSQL(YES).
4. Precompiler options do not affect ODBC behavior.
5. For certain compilers, there is another compiler option that can suppress the CODEPAGE compiler option (for COBOL) to be passed to the DB2 coprocessor. In which case, the source CCSID will be resolved to the CCSID suboption of the SQL compiler option or to the default CCSID from DSNHDECP.
6. The DB2 coprocessor ignores this option when the DB2 coprocessor is invoked by the compiler to prepare the application. However, if you are using PL/I V4.1 or later, it is supported.

Related concepts:

“Precision for operations with decimal numbers” on page 708

 Datetime values (DB2 SQL)

Related tasks:

“Creating a package version” on page 945

“Setting the program level” on page 959

Related reference:

“Defaults for SQL processing options”

Defaults for SQL processing options

Some SQL statement processing options have default values that are based on values that are specified on the DB2I Application Programming Defaults panels.

The following table shows those options and defaults.

Table 151. IBM-supplied installation default SQL statement processing options. The installer can change these defaults.

Install option	Install default	Equivalent SQL statement processing option	Available SQL statement processing options
STRING DELIMITER	quotation mark (")	QUOTE	APOSTQUOTE
SQL STRING DELIMITER	quotation mark (")	QUOTESQL	APOSTSQLQUOTESQL
DECIMAL POINT IS	PERIOD	PERIOD	COMMAPERIOD
DATE FORMAT	ISO	DATE(ISO)	DATE(ISO USA EUR JIS LOCAL)
DECIMAL ARITHMETIC	DEC15	DEC(15)	DEC(15 31)
MIXED DATA	NO	CCSID(n)	CCSID(n)
LANGUAGE DEFAULT	COBOL	HOST(COBOL)	HOST(ASM C[(FOLD)] CPP[(FOLD)] IBMCOB FORTRAN PLI)
STD SQL LANGUAGE	NO	STDSQL(NO)	STDSQL(YES NO 86)
TIME FORMAT	ISO	TIME(ISO)	TIME(IS USA EUR JIS LOCAL)

Table 151. IBM-supplied installation default SQL statement processing options (continued). The installer can change these defaults.

Install option	Install default	Equivalent SQL statement processing option	Available SQL statement processing options
<p>Notes: For dynamic SQL statements, another application programming default, USE FOR DYNAMICRULES, determines whether DB2 uses the application programming default or the SQL statement processor option for the following installation options:</p> <ul style="list-style-type: none"> • STRING DELIMITER • SQL STRING DELIMITER • DECIMAL POINT IS • DECIMAL ARITHMETIC <p>If the value of USE FOR DYNAMICRULES is YES, dynamic SQL statements use the application programming defaults. If the value of USE FOR DYNAMICRULES is NO, dynamic SQL statements in packages or plans with bind, define, and invoke behavior use the SQL statement processor options.</p>			

Some SQL statement processor options have default values based on the host language. Some options do not apply to some languages. The following table shows the language-dependent options and defaults.

Table 152. Language-dependent DB2 precompiler options and defaults

HOST value	Defaults
ASM	APOST ¹ , APOSTSQL ¹ , PERIOD ¹ , TWOPASS, MARGINS(1,71,16)
C or CPP	APOST ¹ , APOSTSQL ¹ , PERIOD ¹ , ONEPASS, MARGINS(1,72)
IBMCOB	QUOTE ² , QUOTESQL ² , PERIOD, ONEPASS ¹ , MARGINS(8,72) ¹
FORTRAN	APOST ¹ , APOSTSQL ¹ , PERIOD ¹ , ONEPASS ¹ , MARGINS(1,72) ¹
PLI	APOST ¹ , APOSTSQL ¹ , PERIOD ¹ , ONEPASS, MARGINS(2,72)
SQL or SQLPL	APOST ¹ , APOSTSQL ¹ , PERIOD ¹ , ONEPASS, MARGINS(1,72)

Notes:

1. Forced for this language; no alternative is allowed.
2. The default is chosen on Application Programming Defaults Panel 1 during installation. The IBM-supplied installation defaults for string delimiters are QUOTE (host language delimiter) and QUOTESQL (SQL escape character). The installer can replace the IBM-supplied defaults with other defaults. The precompiler options that you specify override any defaults that are in effect.

SQL statement processing defaults for dynamic statements

Generally, dynamic statements use the defaults that are specified during installation. However, if the value of application defaults module parameter DYNRULS is NO, you can use these options for dynamic SQL statements in packages or plans with bind, define, or invoke behavior:

- COMMA or PERIOD
- APOST or QUOTE
- APOSTSQL or QUOTESQL
- DEC(15) or DEC(31)

Related concepts:

“DYNAMICRULES bind option” on page 959

SQL options for DRDA access

Certain SQL statement processing options are relevant when you prepare a package to be run with DRDA access.

The following SQL statement processing options are relevant for DRDA access:

CONNECT

Use **CONNECT(2)**, explicitly or by default.

CONNECT(1) causes your **CONNECT** statements to allow only the restricted function known as “remote unit of work”. Be particularly careful to avoid **CONNECT(1)** if your application updates more than one DBMS in a single unit of work.

SQL

Use **SQL(ALL)** explicitly for a package that runs on a server that *is not* DB2 for z/OS. The precompiler then accepts any statement that obeys DRDA rules.

Use **SQL(DB2)**, explicitly or by default, if the server is DB2 for z/OS only. The precompiler then rejects any statement that does not obey the rules of DB2 for z/OS.

Compiling and link-editing an application

If you use the DB2 precompiler, your next step in the program preparation process is to compile and link-edit your program. As with the precompile step, you have a choice of methods.

You can use one of the following methods to compile and link-edit an application:

- DB2I panels
- The DSNH command procedure (a TSO CLIST)
- JCL procedures supplied with DB2
- JCL procedures supplied with a host language compiler

If you use the DB2 coprocessor, you process SQL statements as you compile your program. For programs other than C and C++ programs, you must use JCL procedures when you use the DB2 coprocessor. For C and C++ programs, you can use either JCL procedures or UNIX System Services on z/OS to invoke the DB2 coprocessor.

The purpose of the link-edit step is to produce an executable load module. To enable your application to interface with the DB2 subsystem, you must use a link-edit procedure that builds a load module that satisfies environment-specific requirements.

TSO and batch: Include the DB2 TSO attachment facility language interface module (DSNELI) or DB2 call attachment facility language interface module (DSNALI) or the Universal Language Interface module (DSNULI).

IMS: Include the DB2 IMS (Version 1 Release 3 or later) language interface module (DFSLI000), which contains the DSNHLI entry point. Also, the IMS RESLIB must precede the SDSNLOAD library in the link list, JOBLIB, or STEPLIB concatenations.

IMS and DB2 share a common alias name, DSNHLI, for the language interface module. You must do the following when you concatenate your libraries:

- If you use IMS, be sure to concatenate the IMS library first so that the application program compiles with the correct IMS version of DSNHLI.
- If you run your application program only under DB2, be sure to concatenate the DB2 library first.

| **CICS:** Include the DB2 CICS language interface module (DSNCLI) or the
| Universal Language Interface module (DSNULI). You can link DSNCLI with your
| program in either 24-bit or 31-bit addressing mode (AMODE=31), but DSNULI
| must be linked with your program in 31-bit addressing mode (AMODE=31). If
| your application runs in 31-bit addressing mode, you should link-edit the DSNCLI
| or DSNULI stub to your application with the attributes AMODE=31 and
| RMODE=ANY so that your application can run above the 16-MB line.

You also need the CICS EXEC interface module that is appropriate for the programming language. CICS requires that this module be the first control section (CSECT) in the final load module.

The size of the executable load module that is produced by the link-edit step varies depending on the values that the SQL statement processor inserts into the source code of the program.

Link-editing a batch program: DB2 has language interface routines for each unique supported environment. DB2 requires the IMS language interface routine for DL/I batch. You need to have DFSLI000 link-edited with the application program.

Related concepts:

“Universal language interface” on page 153

Related tasks:

Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915

Related reference:

 DSNH (TSO CLIST) (DB2 Commands)

Related information:

 CICS DB2 program preparation steps (CICS Transaction Server for z/OS)

Binding an application

| You must bind the DBRM that is produced by the SQL statement processor to a
| package before your DB2 application can run.

Each package that you bind can contain only one DBRM.

Exception: You do not need to bind a DBRM if the only SQL statement in the program is SET CURRENT PACKAGESET.

Because you do not need a plan or package to execute the SET CURRENT PACKAGESET statement, the ENCODING bind option does not affect the SET CURRENT PACKAGESET statement. An application that needs to provide a host variable value in an encoding scheme other than the system default encoding scheme must use the DECLARE VARIABLE statement to specify the encoding scheme of the host variable.

You must bind plans locally, regardless of whether they reference packages that run remotely. However, you must bind the packages that run at remote locations at those remote locations.

For C and C++ programs whose corresponding DBRMs are in HFS files, you can use the command line processor to bind the DBRMs to packages. Optionally, you

can also copy the DBRM into a partitioned data set member by using the `oput` and `oget` commands and then bind it by using conventional JCL.

From a DB2 requester, you can run a plan by specifying it in the `RUN` subcommand, but you cannot run a package directly. You must include the package in a plan and then run the plan.

Develop a naming convention and strategy for the most effective and efficient use of your plans and packages.

- To bind a new plan or package, other than a trigger package, use the subcommand `BIND PLAN` or `BIND PACKAGE` with the option `ACTION(REPLACE)`.

To bind a new trigger package, re-create the trigger associated with the trigger package.

Binding a DBRM to a package

You can bind a DBRM to a package and then bind that package to a plan.

When you bind a package, you specify the collection to which the package belongs. The collection is not a physical entity, and you do not create it; the collection name is merely a convenient way of referring to a group of packages.

To bind a package, you must have the proper authorization.

Binding packages at a remote location

When your application accesses data through remote access, you must bind copies of your requester application packages at any location which will be accessed by the application.

If a local stored procedure uses a cursor to access data, and the cursor-related statement is bound in a separate package under the stored procedure, you must bind this separate package both locally and remotely. In addition, the invoker or owner of the stored procedure must be authorized to execute both local and remote packages. At your local requesting system, you must bind a plan whose package list includes all those packages, local and remote.

To bind a package at a remote DB2 system, you must have all the privileges or authority there that you would need to bind the package on your local system. To bind a package at another type of a system, such as DB2 Server for VSE & VM, you need any privileges that the other system requires to execute its SQL statements and use its data objects.

The bind process for a remote package is the same as for a local package, except that the local communications database must be able to recognize the location name that you use as resolving to a remote location.

Example:

To bind the DBRM `PROGA` at the location `PARIS`, in the collection `GROUP1`, use:

```
BIND PACKAGE(PARIS.GROUP1) COPY(GROUP1.PROGA)
```

Then, include the remote package in the package list of a local plan, such as `PLANB`, by using:

```
BIND PLAN (PLANB) PKLIST (*.GROUP1.*)
```

The ENCODING bind option has the following effect on a remote application:

- If you bind a package locally, which is recommended, and you specify the ENCODING bind option for the local package, the ENCODING bind option for the local package applies to the remote application.
- If you do not bind a package locally, and you specify the ENCODING bind option for the plan, the ENCODING bind option for the plan applies to the remote application.
- If you do not specify an ENCODING bind option for the package or plan at the local site, the value of APPLICATION ENCODING that was specified on installation panel DSNTIPF at the local site applies to the remote application.

When you bind or rebind, DB2 checks authorizations, reads and updates the catalog, and creates the package in the directory at the remote site. DB2 does not read or update catalogs or check authorizations at the local site.

If you specify the option EXPLAIN(YES) or EXPLAIN(ONLY) and you do not specify the option SQLERROR(CONTINUE), PLAN_TABLE must exist at the location that is specified on the BIND or REBIND subcommand. This location could also be the default location.

If you bind with the option COPY, the COPY privilege must exist locally. DB2 performs authorization checking, reads and updates the catalog, and creates the package in the directory at the remote site. DB2 reads the catalog records that are related to the copied package at the local site. DB2 converts values that are returned from the remote site in ISO format if all of the following conditions are true:

- If the local site is installed with time or date format LOCAL
- A package is created at a remote site with the COPY option
- The SQL statement does not specify a different format.

After you bind a package, you can rebind, free, or bind it with the REPLACE option using either a local or a remote bind.

Creating a package version

If you want to run different versions of a program without needing to make changes to the associated application plan, use package versions. This technique is useful if you need to make changes to your program without causing an interruption to the availability of the program.

You can create a different package version for each version of the program. Each package has the same package name and collection name, but a different version number is associated with each package. The plan that includes that package includes all versions of that package. Thus, you can run a program that is associated with any one of the package versions without having to rebind the application plan, rename the plan, or change any RUN subcommands that use it.

To create a package version:

1. Precompile your program with the option VERSION(*version-identifier*).
2. Bind the resulting DBRM with the same collection name and package name as any existing versions of that package. When you run the program, DB2 uses the package version that you specified when you precompiled it.

Suppose that you bound a plan with the following statement:

```
BIND PLAN (PLAN1) PKLIST (COLLECT.*)
```

The following steps show how to create two versions of a package, one for each of two programs.

Step number	For package version 1	For package version 2
1	Precompile program 1. Specify VERSION(1).	Precompile program version 2. Specify VERSION(2).
2	Bind the DBRM with the collection name COLLECT and the package name PACKA.	Bind the DBRM with the collection name COLLECT and package name PACKA.
3	Link-edit program 1 into your application.	Link-edit program 2 into your application.
4	Run the application; it uses program 1 and PACKA, VERSION 1.	Run the application; it uses program 2 and PACKA, VERSION 2.

Binding a DBRM that is in an HFS file to a package or collection

If DBRMs are in z/OS UNIX HFS files, you can use the command line processor to bind the DBRMs to packages at the target DB2 server. Optionally, you can also copy the DBRM into a partitioned data set member by using the TSO/E `oput` and `oget` commands and then bind the DBRM by using conventional JCL.

Only DBRMs for C and C++ programs can be generated to HFS files.

Restrictions:

You cannot specify the `REBIND` command with the command line processor. Alternatively, specify the `BIND` command with the `ACTION(REPLACE)` option.

You cannot specify the `FREE PACKAGE` command with the command line processor. Alternatively, specify the `DROP PACKAGE` statement to drop the existing packages.

To bind a DBRM that is in an HFS file to a package or collection:

1. Invoke the command line processor and connect to the target DB2 server.
2. Specify the `BIND` command with the appropriate options.

Related concepts:

 [Command line processor \(DB2 Commands\)](#)

Related tasks:

“Processing SQL statements by using the DB2 coprocessor” on page 926

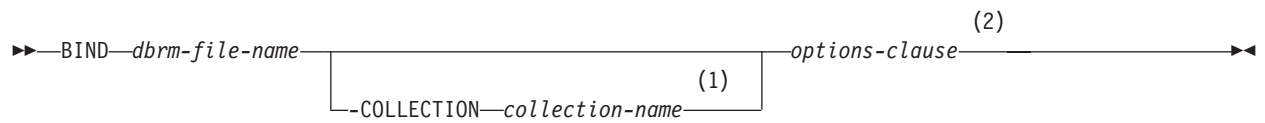
Related reference:

“Command line processor `BIND` command”

Command line processor `BIND` command:

Use the command line processor `BIND` command to bind DBRMs that are in z/OS UNIX HFS files to packages.

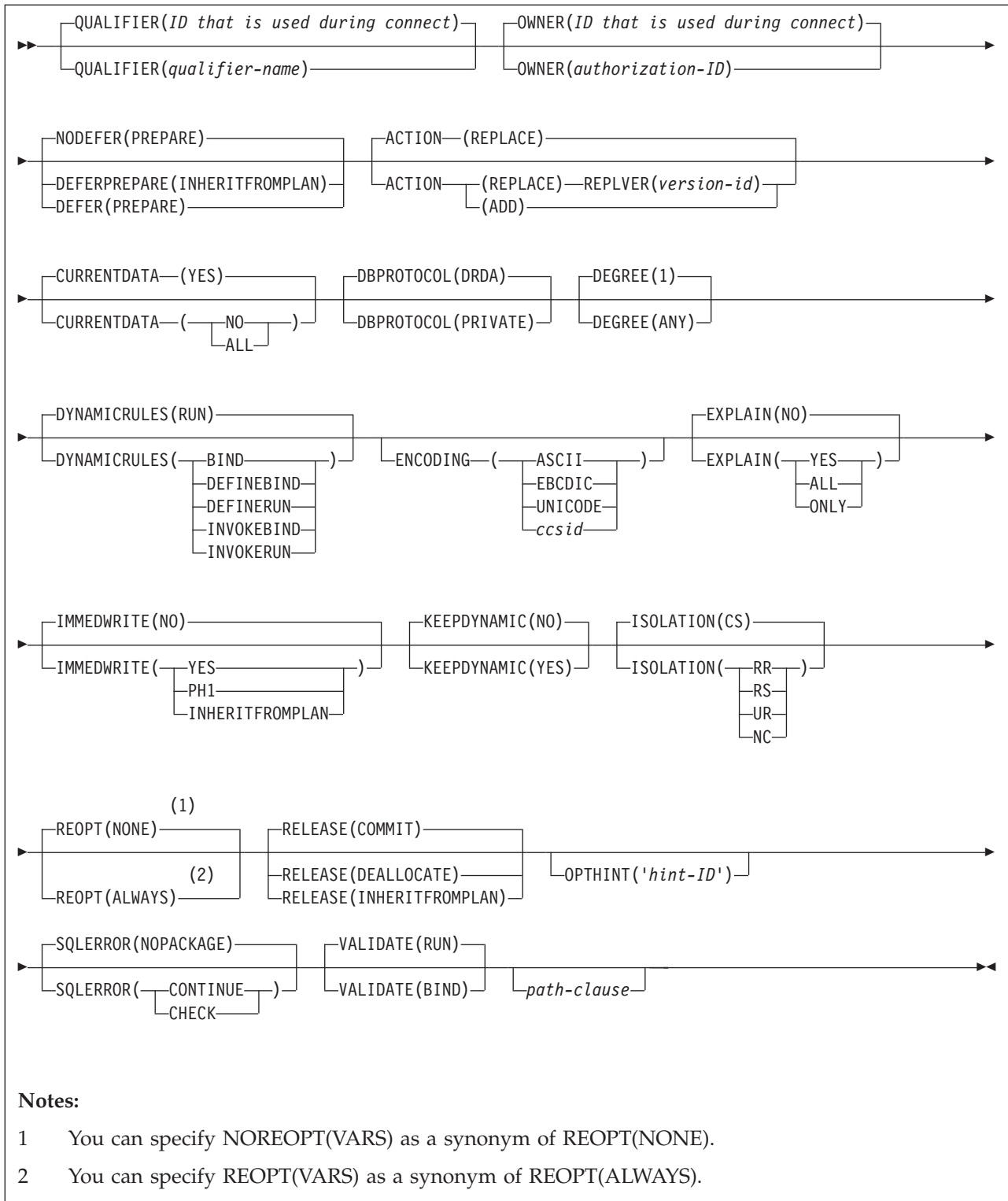
The following diagram shows the syntax for the command line processor `BIND` command.



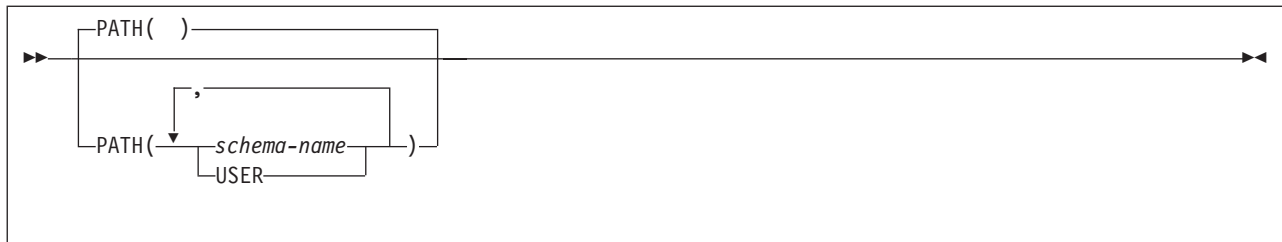
Notes:

- 1 If you do not specify a collection, DB2 uses NULLID.
- 2 You can specify the options after *collection-name* in any order.

options-clause:



path-clause:



The following options are unique to this diagram:

CURRENTDATA (ALL)

Specifies that for all cursors data currency is required and block fetching is inhibited.

SQLERROR (CHECK)

Specifies that the command line processor is to only check for SQL errors in the DBRM. No package is generated.


IMMEDWRITE (PH1)

Specifies that normal write activity is done. This option is equivalent to IMMEDIATEWRITE(NO).

EXPLAIN (ALL)

Specifies that DB2 is to insert information into the appropriate EXPLAIN tables. This option is equivalent to EXPLAIN (YES).

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

Binding an application plan

An application plan can include package lists.

To bind an application plan, use the BIND PLAN subcommand with at least one of the following options:

MEMBER

| Specify this option to bind DBRMs to a package and then bind the package list
 | to a plan. After the keyword MEMBER, specify the member names of the
 | DBRMS.

PKLIST

Specify this option to include package lists in the plan. After the keyword PKLIST, specify the names of the packages to include in the package list. To include an entire collection of packages in the list, use an asterisk after the collection name. For example, PKLIST(GROUP1.*).

The resulting plan consists of the following information:

- Any programs that are associated with DBRMs in the MEMBER list
- Any programs that are associated with packages and collections that are identified in PKLIST

Specifying the package list for the PKLIST option of BIND PLAN:

The order in which you specify packages in a package list can affect run time performance. Searching for the specific package involves searching the DB2

directory, which can be costly. When you use *collection-id.** with the PKLIST keyword, you should specify first the collections in which DB2 is most likely to find a package.

For example, assume that you perform the following bind:

```
BIND PLAN (PLAN1) PKLIST (COLL1.*, COLL2.*, COLL3.*, COLL4.*)
```

Then you execute program PROG1. DB2 does the following package search:

1. Checks to see if program PROG1 is bound as part of the plan
2. Searches for COLL1.PROG1.*timestamp*
3. If it does not find COLL1.PROG1.*timestamp*, searches for COLL2.PROG1.*timestamp*
4. If it does not find COLL2.PROG1.*timestamp*, searches for COLL3.PROG1.*timestamp*
5. If it does not find COLL3.PROG1.*timestamp*, searches for COLL4.PROG1.*timestamp*.

When both special registers CURRENT PACKAGE PATH and CURRENT PACKAGESET contain an empty string: If you do not set these special registers, DB2 searches for a DBRM or a package in one of these sequences:

- At the local location (if CURRENT SERVER is blank or specifies that location explicitly), the order is:
 1. All packages that are already allocated to the plan while the plan is running.
 2. All unallocated packages that are explicitly specified in, and all collections that are completely included in, the package list of the plan. DB2 searches for packages in the order that they appear in the package list.
- At a remote location, the order is:
 1. All packages that are already allocated to the plan at that location while the plan is running.
 2. All unallocated packages that are explicitly specified in, and all collections that are completely included in, the package list of the plan, whose locations match the value of CURRENT SERVER. DB2 searches for packages in the order that they appear in the package list.

If you use the BIND PLAN option DEFER(PREPARE), DB2 does not search all collections in the package list.

If the order of search is not important: In many cases, the order in which DB2 searches the packages is not important to you and does not affect performance. For an application that runs only at your local DB2 system, you can name every package differently and include them all in the same collection. The package list on your BIND PLAN subcommand can read:

```
PKLIST (collection.*)
```

You can add packages to the collection even after binding the plan. DB2 lets you bind packages having the same package name into the same collection only if their version IDs are different.

If your application uses DRDA access, you must bind some packages at remote locations. Use the same collection name at each location, and identify your package list as:

```
PKLIST (*.collection.*)
```


If you use an asterisk for part of a name in a package list, DB2 checks the authorization for the package to which the name resolves at run time. To avoid the checking at run time in the preceding example, you can grant EXECUTE authority for the entire collection to the owner of the plan before you bind the plan.

Related tasks:

[➤ Improving performance for applications that access distributed data \(DB2 Performance\)](#)

Related reference:

[➤ BIND PLAN \(DSN\) \(DB2 Commands\)](#)

[➤ CURRENT PACKAGE PATH \(DB2 SQL\)](#)

[➤ CURRENT PACKAGESET \(DB2 SQL\)](#)

How DB2 identifies packages at run time

The DB2 precompiler or DB2 coprocessor identifies each call to DB2 with a consistency token. The same consistency token identifies the DBRM that the SQL statement processor produces and the package to which you bound the DBRM.

When you run the program, DB2 uses the consistency token in matching the call to DB2 to the correct DBRM. Usually, the consistency token is in an internal DB2 format. You can override that token if you want.

You also need other identifiers. The consistency token alone does not necessarily identify a unique package. You can bind the same DBRM to many packages, at different locations and in different collections, and you can include all those packages in the package list of the same plan. All those packages will have the same consistency token. You can specify a particular location or a particular collection at run time.

Related tasks:

“Setting the program level” on page 959

Specifying the location of the package that DB2 is to use

When your program executes SQL statements, DB2 uses the value in the CURRENT SERVER special register to determine the location of the necessary package. If the current server is your local DB2 subsystem and it does not have a location name, the value in the special register is blank.

You can change the value of CURRENT SERVER by using the SQL CONNECT statement in your program. If you do not use CONNECT, the value of CURRENT SERVER is the location name of your local DB2 subsystem (or blank, if your DB2 subsystem has no location name).

Specifying the package collection that DB2 is to use

To ensure that DB2 uses the intended package collection and does not waste time searching, explicitly specify the package collection that you want DB2 to use.

You can use the special register CURRENT PACKAGE PATH or CURRENT PACKAGESET (if CURRENT PACKAGE PATH is not set) to specify the collections that are to be used for package resolution. The CURRENT PACKAGESET special register contains the name of a single collection, and the CURRENT PACKAGE PATH special register contains a list of collection names.

If you do not set these registers, they contain an empty string when your application begins to run, and they remain as an empty string. In this case, DB2 searches the available collections.

However, explicitly specifying the intended collection by using the special registers can avoid a potentially costly search through a package list that has many qualifying entries. In addition, DB2 uses the values in these special registers for applications that do not run under a plan.

When you call a stored procedure, the special register CURRENT PACKAGESET contains the value that you specified for the COLLID parameter when you defined the stored procedure. If the routine was defined without a value for the COLLID parameter, the value of the special register is inherited from the calling program. Also, the special register CURRENT PACKAGE PATH contains the value that you specified for the PACKAGE PATH parameter when you defined the stored procedure. When the stored procedure returns control to the calling program, DB2 restores this register to the value that it contained before the call.

Related tasks:

“Binding an application plan” on page 949

“Overriding the values that DB2 uses to resolve package lists”

Overriding the values that DB2 uses to resolve package lists

DB2 resolves package lists by searching the available collections in a particular order. To avoid this search, you can specify the values that DB2 should use for package resolution.

If you set the special register CURRENT PACKAGE PATH or CURRENT PACKAGESET, DB2 skips the check for programs that are part of a plan and uses the values in these registers for package resolution.

If you set CURRENT PACKAGE PATH, DB2 uses the value of CURRENT PACKAGE PATH as the collection name list for package resolution. For example, if CURRENT PACKAGE PATH contains the list COLL1, COLL2, COLL3, COLL4, DB2 searches for the first package that exists in the following order:

COLL1.PROG1.*timestamp*
COLL2.PROG1.*timestamp*
COLL3.PROG1.*timestamp*
COLL4.PROG1.*timestamp*

If you set CURRENT PACKAGESET and **not** CURRENT PACKAGE PATH, DB2 uses the value of CURRENT PACKAGESET as the collection for package resolution. For example, if CURRENT PACKAGESET contains COLL5, DB2 uses COLL5.PROG1.*timestamp* for the package search.

When CURRENT PACKAGE PATH is set, the server that receives the request ignores the collection that is specified by the request and instead uses the value of CURRENT PACKAGE PATH at the server to resolve the package. Specifying a collection list with the CURRENT PACKAGE PATH special register can avoid the need to issue multiple SET CURRENT PACKAGESET statements to switch collections for the package search.

The following table shows examples of the relationship between the CURRENT PACKAGE PATH special register and the CURRENT PACKAGESET special register.

Table 153. Scope of CURRENT PACKAGE PATH

Example	What happens
SET CURRENT PACKAGESET SELECT ... FROM T1 ...	The collection in PACKAGESET determines which package is invoked.
SET CURRENT PACKAGE PATH SELECT ... FROM T1 ...	The collections in PACKAGE PATH determine which package is invoked.
SET CURRENT PACKAGESET SET CURRENT PACKAGE PATH SELECT ... FROM T1 ...	The collections in PACKAGE PATH determine which package is invoked.
SET CURRENT PACKAGE PATH CONNECT TO S2 ... SELECT ... FROM T1 ...	PACKAGE PATH at server S2 is an empty string because it has not been explicitly set. The values from the PKLIST bind option of the plan that is at the requester determine which package is invoked. ¹
SET CURRENT PACKAGE PATH = 'A,B' CONNECT TO S2 ... SET CURRENT PACKAGE PATH = 'X,Y' SELECT ... FROM T1 ...	The collections in PACKAGE PATH that are set at server S2 determine which package is invoked.
SET CURRENT PACKAGE PATH SELECT ... FROM S2.QUAL.T1 ...	Three-part table name. On implicit connection to server S2, PACKAGE PATH at server S2 is inherited from the local server. The collections in PACKAGE PATH at server S2 determine which package is invoked.

Notes:

1. When CURRENT PACKAGE PATH is set at the requester (and not at the remote server), DB2 passes one collection at a time from the list of collections to the remote server until a package is found or until the end of the list. Each time a package is not found at the server, DB2 returns an error to the requester. The requester then sends the next collection in the list to the remote server.

Bind process for remote access

You can use different bind processes to access data at a remote server.

Example

Suppose that CHIEMP is an alias for table CHICAGO.DSN8A10.EMP. Access data at a remoter server by using the following query:

```
SELECT * FROM CHIEMP
WHERE EMPNO = '0001000';
```

If you bind the DBRM that contains the statement by using the following process, you access the server using remote access:

Local-bind DRDA access process:

1. Bind the DBRM into a package at the local DB2.
2. Bind package copy at the CHICAGO test site.
3. When the application is completed testing, bind DBRM into a package at the local production site.
4. Bind package copy to the CHICAGO production site.

Example

Suppose that you need to access data at a remote server CHICAGO, by using the following SELECT statements:

```
EXEC SQL SELECT * FROM CHIEMP  
WHERE EMPNO = '0001000';
```

where CHIEMP is an alias for table CHICAGO.DSN8A10.EMP.

Suppose that the query is bound locally. You need to BIND PACKAGE COPY the query to the remote server site. Now that you have both a local and a remote package, you must have a plan that has both the local and remote packages in the package list.

Example

Suppose that you need to call a stored procedure at the remote server ATLANTA, by using the following CONNECT and CALL statements:

```
EXEC SQL  
CONNECT TO ATLANTA;  
EXEC SQL  
CALL procedure_name (parameter_list);
```

The parameter list is a list of host variables that is passed to the stored procedure and into which it returns the results of its execution. To execute, the stored procedure must already exist at the ATLANTA server.

Bind options for remote access

Binding a package to run at a remote location is like binding a package to run at your local DB2 subsystem. Binding a plan to run the package is like binding any other plan. However, a few differences exist.

For the general instructions, see Chapter 17, "Preparing an application to run on DB2 for z/OS," on page 915.

BIND PLAN options for DRDA access

The following options of BIND PLAN are particularly relevant to binding a plan that uses DRDA access:

DISCONNECT

For most flexibility, use DISCONNECT(EXPLICIT), explicitly or by default. That requires you to use RELEASE statements in your program to explicitly end connections.

The other values of the option are also useful.

DISCONNECT(AUTOMATIC) ends all remote connections during a commit operation, without the need for RELEASE statements in your program.

DISCONNECT(CONDITIONAL) ends remote connections during a commit operation except when an open cursor defined as WITH HOLD is associated with the connection.

SQLRULES

Use SQLRULES(DB2), explicitly or by default.

SQLRULES(STD) applies the rules of the SQL standard to your **CONNECT** statements, so that **CONNECT TO x** is an error if you are already connected to *x*. Use **STD** only if you want that statement to return an error code.

If your program selects LOB data from a remote location, and you bind the plan for the program with **SQLRULES(DB2)**, the format in which you retrieve the LOB data with a cursor is restricted. After you open the cursor to retrieve the LOB data, you must retrieve all of the data using a LOB variable, or retrieve all of the data using a LOB locator variable. If the value of **SQLRULES** is **STD**, this restriction does not exist.

If you intend to switch between LOB variables and LOB locators to retrieve data from a cursor, execute the **SET SQLRULES=STD** statement before you connect to the remote location.

CURRENTDATA

Use **CURRENTDATA(NO)** to force block fetch for ambiguous cursors.

ENCODING

Use this option to control the encoding scheme that is used for static SQL statements in the plan and to set the initial value of the **CURRENT APPLICATION ENCODING SCHEME** special register.

For applications that execute remotely and use explicit **CONNECT** statements, **DB2** uses the **ENCODING** value for the plan. For applications that execute remotely and use implicit **CONNECT** statements, **DB2** uses the **ENCODING** value for the package that is at the site where a statement executes.

BIND PACKAGE options for DRDA access

The following options of **BIND PACKAGE** are relevant to binding a package to be run using **DRDA** access:

location-name

Name the location of the server at which the package runs.

The privileges needed to run the package must be granted to the owner of the package at the server. If you are not the owner, you must also have **SYSCTRL** authority or the **BINDAGENT** privilege that is granted locally.

SQLERROR

Use **SQLERROR(CONTINUE)** if you used **SQL(ALL)** when precompiling. That creates a package even if the bind process finds SQL errors, such as statements that are valid on the remote server but that the precompiler did not recognize. Otherwise, use **SQLERROR(NOPACKAGE)**, explicitly or by default.

CURRENTDATA

Use **CURRENTDATA(NO)** to force block fetch for ambiguous cursors.

OPTIONS

When you make a remote copy of a package using **BIND PACKAGE** with the **COPY** option, use this option to control the default bind options that **DB2** uses. Specify:

COMPOSITE to cause **DB2** to use any options you specify in the **BIND PACKAGE** command. For all other options, **DB2** uses the options of the copied package. **COMPOSITE** is the default.

COMMAND to cause **DB2** to use the options you specify in the **BIND PACKAGE** command. For all other options, **DB2** uses the defaults for the server on which the package is bound. This helps ensure that the server supports the options with which the package is bound.

ENCODING

Use this option to control the encoding scheme that is used for static SQL statements in the package and to set the initial value of the CURRENT APPLICATION ENCODING SCHEME special register.

The default ENCODING value for a package that is bound at a remote DB2 for z/OS server is the system default for that server. The system default is specified at installation time in the APPLICATION ENCODING field of panel DSNTIPF.

For applications that execute remotely and use explicit CONNECT statements, DB2 uses the ENCODING value for the plan. For applications that execute remotely and use implicit CONNECT statements, DB2 uses the ENCODING value for the *package* that is at the site where a statement executes.

Related concepts:

 Bind options for locks (DB2 Performance)

Related tasks:

 BIND options for distributed applications (DB2 Performance)

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

Checking which BIND PACKAGE options a particular server supports

You can request only the options of the BIND PACKAGE command that are supported by the server by specifying those options at the requester.

To find out which options are supported by a specific server DBMS, refer to the documentation provided for that server.

For specific DB2 bind information, refer to the following documentation:

- For guidance in using DB2 bind options and performing a bind process, see Chapter 17, “Preparing an application to run on DB2 for z/OS,” on page 915.
- For the syntax of DB2 BIND command, see the topics BIND PACKAGE (DSN) (DB2 Commands) and BIND PLAN (DSN) (DB2 Commands).
- For the syntax of DB2 REBIND command, see the topics REBIND PACKAGE (DSN) (DB2 Commands) and REBIND PLAN (DSN) (DB2 Commands).

Binding a batch program

Before a batch program can issue SQL statements, a DB2 plan must exist.

The owner of the plan or package must have all the privileges that are required to execute the SQL statements embedded in it.

You can specify the plan name to DB2 in one of the following ways:

- In the DDITV02 input data set.
- In subsystem member specification.
- By default; the plan name is then the application load module name that is specified in DDITV02.

DB2 passes the plan name to the IMS attach package. If you do not specify a plan name in DDITV02, and a resource translation table (RTT) does not exist or the name is not in the RTT, DB2 uses the passed name as the plan name. If the name exists in the RTT, the name translates to the plan that is specified for the RTT.

Recommendation: Give the DB2 plan the same name as that of the application load module, which is the IMS attachment facility default. The plan name must be the same as the program name.

Conversion of DBRMs that are bound to a plan to DBRMs that are bound to a package

You must bind all DBRMs into a package, and bind the packages into a plan. One package can have only one DBRM.

The default REBIND PLAN COLLID (*) option converts all plans with DBRMs into plans with a package list. You can use this technique for local applications only. If the plan that you specify already contains both DBRMs and package lists, the newly converted package entries will be inserted into the front of the existing package list. Subsystem parameter DISALLOW_DEFAULT_COLLID must be set to NO for the rebind to be successful.

Example: converting all plans

The following examples converts all DBRMs that are bound with plan X into packages under collection ID: DSN_DEFAULT_COLLID_X.

```
REBIND PLAN(X) COLLID(*);
```

Example: specifying a collection ID

The following examples converts DBRMs that are bound with plan X into packages under the *my_collection* collection ID.

```
REBIND PLAN(x) COLLID('my_collection');
```

Example: rebinding multiple plans which may contain DBRMs

In the following example, BIND will traverse through each plan that is specified in the REBIND PLAN command statement and will convert the DBRMs accordingly, and until none of the DBRMs are bound with plans.

```
REBIND PLAN (X1, X2, X3) COLLID (collection_id|*);
```

Example: rebinding all plans which may contain DBRMs

In the following example, BIND will traverse through all plans that are specified in the SYSPLAN table and will convert the DBRMs accordingly, and until none of the DBRMs are bound with plans.

```
REBIND PLAN (*) COLLID (collection_id|*);
```

Example: specifying a package list

The following examples converts all DBRMs that are bound with plan X into packages under collection ID: DSN_DEFAULT_COLLID_X.

- If plan X does not have a package list, the newly converted package entries will be appended to the front of package list Z and then package list Z will be added to plan X.
- If plan X has both a package list and DBRMs, the newly converted package entries will be appended to the front of package list Z and then package list Z will replace the existing package list.
- If plan X has only a package list, then package list Z will replace the existing package list.


```
REBIND PLAN (x) COLLID (collection_id|*) PKLIST(Z);
```

Example: specifying no package list

The following examples converts all DBRMs that are bound with plan X into packages under collection ID: DSN_DEFAULT_COLLID_X.

- If plan X has both a package list and DBRMs, the existing package list will be deleted, and the new package list will be bound into plan X.
- If plan X has only DBRMs, the DBRMs will be converted into packages accordingly and added to plan X. The NOPKLIST option will be ignored.
- If plan X does not have DBRMs, then the existing package list, if any, will be deleted.

```
REBIND PLAN (x) COLLID (collection_id|*) NOPKLIST;
```

Converting an existing plan into packages to run remotely

If you have an existing application that you want to run at a remote location by using remote access, you need a new plan that includes those remote packages in its package list.

To turn an existing plan with member DBRMs into packages to run remotely, perform the following actions for each remote location:

1. Choose a name for a collection to contain member DBRMs, such as REMOTE1.
2. Convert the plan into a plan with a package list of packages. Subsystem parameter DISALLOW_DEFAULT_COLLID must be set to NO for the rebind to be successful.

```
REBIND PLAN(REMOTE1)COLLID(*)
```

Specifying COLLID(*) produces the packages under the collection of DSN_DEFAULT_COLLID_planname.

3. Query SYSIBM.SYSPACKDEP, to see if any of the packages have a dependency on an alias. That alias is a definition for a 3-part name.
 - a. For each of the packages that have a dependency on an alias:

```
  BIND PACKAGE(location.remote_server_collid)
  COPY(DSN_DEFAULT_COLLID_planname.pgkid)
  COPYVER(...) OPTIONS(COMPOSITE)
```

4. Adjust the location's package list. If prior to this process, the plan had no package list, after 2 it will have a package list containing DSN_DEFAULT_COLLID_planname.pgkid.

```
REBIND PLAN PKLIST
(*.DSN_DEFAULT_COLLID_planname.pgkid* *.remote_server_collid.* )
```

When you now run the existing application at your local DB2 system using the new application plan, these things happen:

- You connect immediately to the remote location that is named in the CURRENTSERVER option.
- DB2 searches for the package in the collection REMOTE1 at the remote location.
- Any UPDATE, DELETE, or INSERT statements in your application affect tables at the remote location.
- Any results from SELECT statements are returned to your existing application program, which processes them as though they came from your local DB2 system.

Setting the program level

The program level defines the level for a particular module. This information is stored in the consistency token, which is in an internal DB2 format. Overriding the program level in the consistency token is possible, if needed, but generally not recommended.

To override the construction of the consistency token by DB2:

Use the LEVEL (*aaaa*) option. DB2 uses the value that you choose for *aaaa* to generate the consistency token. Although this method is not recommended for general use and the DSNH CLIST or the DB2 Program Preparation panels do not support it, this method enables you to perform the following actions:

1. Change the source code (but not the SQL statements) in the DB2 precompiler output of a bound program.
2. Compile and link-edit the changed program.
3. Run the application without rebinding a plan or package.

DYNAMICRULES bind option

The DYNAMICRULES bind option and the run time environment determine the values for the dynamic SQL attributes.

The BIND or REBIND option DYNAMICRULES determines what values apply at run time for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that DB2 uses to parse and semantically verify dynamic SQL statements
- Whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements

In addition, the run time environment of a package controls how dynamic SQL statements behave at run time. The two possible run time environments are:

- The package runs as part of a stand-alone program.
- The package runs as a stored procedure or user-defined function package, or it runs under a stored procedure or user-defined function.

A package that runs under a stored procedure or user-defined function is a package whose associated program meets one of the following conditions:

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

Dynamic SQL statement behavior:

The dynamic SQL attributes that are determined by the value of the DYNAMICRULES bind option and the run time environment are collectively called the *dynamic SQL statement behavior*. The four behaviors are:

- Run behavior
- Bind behavior
- Define behavior
- Invoke behavior

The following table shows the combination of DYNAMICRULES value and run time environment that yield each dynamic SQL behavior.

Table 154. How DYNAMICRULES and the run time environment determine dynamic SQL statement behavior

DYNAMICRULES value	Behavior of dynamic SQL statements in a stand-alone program environment	Behavior of dynamic SQL statements in a user-defined function or stored procedure environment
BIND	Bind behavior	Bind behavior
RUN	Run behavior	Run behavior
DEFINEBIND	Bind behavior	Define behavior
DEFINERUN	Run behavior	Define behavior
INVOKEBIND	Bind behavior	Invoke behavior
INVOKERUN	Run behavior	Invoke behavior

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

The following table shows the dynamic SQL attribute values for each type of dynamic SQL behavior.

Table 155. Definitions of dynamic SQL statement behaviors

Dynamic SQL attribute	Setting for dynamic SQL attributes			
	Bind behavior	Run behavior	Define behavior	Invoke behavior
Authorization ID	Plan or package owner	Current SQLID	User-defined function or stored procedure owner	Authorization ID of invoker ¹
Default qualifier for unqualified objects	Bind OWNER or QUALIFIER value	CURRENT SCHEMA	User-defined function or stored procedure owner	Authorization ID of invoker
CURRENT SQLID ²	Not applicable	Applies	Not applicable	Not applicable
Source for application programming options	Determined by DSNHDECP or a user-specified application defaults module parameter DYNRULS ³	Install panel DSNTIP4	Determined by DSNHDECP or a user-specified application defaults module parameter DYNRULS ³	Determined by DSNHDECP or a user-specified application defaults module parameter DYNRULS ³
Can execute GRANT, REVOKE, CREATE, ALTER, DROP, RENAME?	No	Yes	No	No

Table 155. Definitions of dynamic SQL statement behaviors (continued)

Dynamic SQL attribute	Setting for dynamic SQL attributes			
	Bind behavior	Run behavior	Define behavior	Invoke behavior
Notes:				
1. If the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs are also checked if they are needed for the required authorization. Otherwise, only one ID, the ID of the invoker, is checked for the required authorization.				
2. DB2 uses the value of CURRENT SQLID as the authorization ID for dynamic SQL statements only for plans and packages that have run behavior. For the other dynamic SQL behaviors, DB2 uses the authorization ID that is associated with each dynamic SQL behavior, as shown in this table.				
The value to which CURRENT SQLID is initialized is independent of the dynamic SQL behavior. For stand-alone programs, CURRENT SQLID is initialized to the primary authorization ID.				
You can execute the SET CURRENT SQLID statement to change the value of CURRENT SQLID for packages with any dynamic SQL behavior, but DB2 uses the CURRENT SQLID value only for plans and packages with run behavior.				
3. The value of DSNHDECP or a user-specified application defaults module parameter DYNRULS, which you specify in field USE FOR DYNAMICRULES in installation panel DSNTIP4, determines whether DB2 uses the SQL statement processing options or the application programming defaults for dynamic SQL statements. See "Options for SQL statement processing" on page 932 for more information.				

Dynamic plan selection

It is beneficial to use dynamic plan selection and packages together. You can convert individual programs in an application that contains many programs and plans, one at a time, to use a combination of plans and packages. This process reduces the number of plans per application; having fewer plans reduces the effort that is needed to maintain the dynamic plan exit routine.

CICS You can use packages and dynamic plan selection together, but when you dynamically switch plans, the following conditions must exist:

- All special registers, including CURRENT PACKAGESET, must contain their initial values.
- The value in the CURRENT DEGREE special register cannot have changed during the current transaction.

Assume that you develop the following programs and DBRMs:

Table 156. Example programs and DBRMs

Program Name	DBRM Name
MAIN	MAIN
PROGA	PLANA
PROGB	PKGB
PROGC	PLANC

You could create packages using the following bind statement:

```
BIND PACKAGE(PKGB) MEMBER(PKGB)
```

The following scenario illustrates thread association for a task that runs program MAIN. Suppose that you execute the following SQL statements in the indicated order. For each SQL statement, the resulting event is described.

1. EXEC CICS START TRANSID(MAIN)

- TRANSID(MAIN) executes program MAIN.
2. EXEC SQL SELECT...
Program MAIN issues an SQL SELECT statement. The default dynamic plan exit routine selects plan MAIN.
 3. EXEC CICS LINK PROGRAM(PROGA)
Program PROGA is invoked.
 4. EXEC SQL SELECT...
DB2 does not call the default dynamic plan exit routine, because the program does not issue a sync point. The plan is MAIN.
 5. EXEC CICS LINK PROGRAM(PROGB)
Program PROGB is invoked.
 6. EXEC SQL SELECT...
DB2 does not call the default dynamic plan exit routine, because the program does not issue a sync point. The plan is MAIN and the program uses package PKGB.
 7. EXEC CICS SYNCPOINT
DB2 calls the dynamic plan exit routine when the next SQL statement executes.
 8. EXEC CICS LINK PROGRAM(PROGC)
Program PROGC is invoked.
 9. EXEC SQL SELECT...
DB2 calls the default dynamic plan exit routine and selects PLANC.
 10. EXEC SQL SET CURRENT SQLID = 'ABC'
The CURRENT SQLID special register is assigned the value 'ABC.'
 11. EXEC CICS SYNCPOINT
DB2 does not call the dynamic plan exit routine when the next SQL statement executes because the previous statement modifies the special register CURRENT SQLID.
 12. EXEC CICS RETURN
Control returns to program PROGB.
 13. EXEC SQL SELECT...

CICS With packages, you probably do not need dynamic plan selection and its accompanying exit routine. A package that is listed within a plan is not accessed until it is executed. However, you can use dynamic plan selection and packages together, which can reduce the number of plans in an application and the effort to maintain the dynamic plan exit routine.

Rebinding an application

You need to rebind an application if you want to change any bind options. You also need to rebind an application when you make changes that affect the plan or package, such as creating an index, but you have not changed the SQL statements.

In some cases, DB2 automatically rebinds the plan or package for you.

If you change the SQL statements, you need to replace the plan or package.

Rebinding a package

You need to rebind a package when you make changes that affect the package but that do not involve changes to the SQL statements. For example, if you create a new index, you need to rebind the package. If you change the SQL, you need to use the BIND PACKAGE command with the ACTION(REPLACE) option.

To rebind a package, other than a trigger package, use the REBIND subcommand. To rebind a trigger package, use the REBIND TRIGGER PACKAGE subcommand. You can change any of bind options for a package when you rebind it.

The following table clarifies which packages are bound, depending on how you specify *collection-id* (coll-id), *package-id* (pkg-id), and *version-id* (ver-id) on the REBIND PACKAGE subcommand.

REBIND PACKAGE does not apply to packages for which you do not have the BIND privilege. An asterisk (*) used as an identifier for collections, packages, or versions does not apply to packages at remote sites.

Table 157. Behavior of REBIND PACKAGE specification. "All" means all collections, packages, or versions at the local DB2 server for which the authorization ID that issues the command has the BIND privilege.

Input	Collections affected	Packages affected	Versions affected
*	all	all	all
.(*)	all	all	all
.	all	all	all
.(ver-id)	all	all	ver-id
.()	all	all	empty string
coll-id.*	coll-id	all	all
coll-id.*(*)	coll-id	all	all
coll-id.*(ver-id)	coll-id	all	ver-id
coll-id.*()	coll-id	all	empty string
coll-id.pkg-id(*)	coll-id	pkg-id	all
coll-id.pkg-id	coll-id	pkg-id	empty string
coll-id.pkg-id()	coll-id	pkg-id	empty string
coll-id.pkg-id.(ver-id)	coll-id	pkg-id	ver-id
.pkg-id()	all	pkg-id	all
*.pkg-id	all	pkg-id	empty string
*.pkg-id()	all	pkg-id	empty string
*.pkg-id.(ver-id)	all	pkg-id	ver-id

Example: The following example shows the options for rebinding a package at the remote location. The location name is SNTERSA. The collection is GROUP1, the

package ID is PROGA, and the version ID is V1. The connection types shown in the REBIND subcommand replace connection types that are specified on the original BIND subcommand.

```
REBIND PACKAGE(SNTERSA.GROUP1.PROGA.(V1)) ENABLE(CICS,REMOTE)
```

You can use the asterisk on the REBIND subcommand for local packages, but not for packages at remote sites. Any of the following commands rebinds all versions of all packages in all collections, at the local DB2 system, for which you have the BIND privilege.

```
REBIND PACKAGE (*)
REBIND PACKAGE (*.*)
REBIND PACKAGE (*.*.(*))
```


Either of the following commands rebinds all versions of all packages in the local collection LEDGER for which you have the BIND privilege.

```
REBIND PACKAGE (LEDGER.*)
REBIND PACKAGE (LEDGER.*(.*))
```

Either of the following commands rebinds the empty string version of the package DEBIT in all collections, at the local DB2 system, for which you have the BIND privilege.


```
REBIND PACKAGE (*.DEBIT)
REBIND PACKAGE (*.DEBIT.())
```

Related tasks:

 [Reusing and comparing access paths at bind and rebind \(DB2 Performance\)](#)

Related reference:

 [BIND and REBIND options for packages and plans \(DB2 Commands\)](#)

 [REBIND PACKAGE \(DSN\) \(DB2 Commands\)](#)

Rebinding a plan

You need to rebind a plan when you make a change to one of the attributes of the plan, such as the package list.

To rebind a plan use the REBIND subcommand. You can change any of bind options for that plan.

When you rebind a plan, use the PKLIST keyword to replace any previously specified package list. Omit the PKLIST keyword to use of the previous package list for rebinding. Use the NOPKLIST keyword to delete any package list that was specified when the plan was previously bound.


Example: Rebinds PLANA and changes the package list:

```
REBIND PLAN(PLANA) PKLIST(GROUP1.*) MEMBER(ABC)
```

Example: Rebinds the plan and drops the entire package list:

```
REBIND PLAN(PLANA) NOPKLIST
```

Related reference:

 [BIND and REBIND options for packages and plans \(DB2 Commands\)](#)

Rebinding lists of plans and packages

In some situations, you need to rebind a set of plans or packages that cannot be described by using asterisks. For example, if a rebind operation terminates, you can generate a rebind subcommand for each object that was not bound.

One situation in which this technique is useful is to complete a rebind operation that has terminated due to lack of resources. A rebind for many objects, such as REBIND PACKAGE (*) for an ID with SYSADM authority, terminates if a needed resource becomes unavailable. As a result, some objects are successfully rebound and others are not. If you repeat the subcommand, DB2 attempts to rebind all the objects again. But if you generate a rebind subcommand for each object that was not rebound, and issue those subcommands, DB2 does not repeat any work that was already done and is not likely to run out of resources.

For a description of the technique and several examples of its use, see “Sample program to create REBIND subcommands for lists of plans and packages.”

Generating lists of REBIND commands

To generate a list of REBIND subcommands for a set of packages that cannot be described, use asterisks, and use information in the DB2 catalog. You can then issue the list of subcommands through DSN.

The following list is an overview of the procedures for REBIND PACKAGE:

1. Use DSNTIAUL to generate the REBIND PACKAGE subcommands for the selected packages.
2. Use DSNTEDIT CLIST to delete extraneous blanks from the REBIND PACKAGE subcommands.
3. Use TSO edit commands to add DSN commands to the sequential data set.
4. Use DSN to execute the REBIND PACKAGE subcommands for the selected packages.

Sample program to create REBIND subcommands for lists of plans and packages

If you cannot use asterisks to identify a list of packages or plans that you want to rebind, you might be able to create the needed REBIND subcommands automatically, by using the sample program DSNTIAUL.

One situation in which this technique might be useful is when a resource becomes unavailable during a rebind of many plans or packages. DB2 normally terminates the rebind and does not rebind the remaining plans or packages. Later, however, you might want to rebind only the objects that remain to be rebound. You can build REBIND subcommands for the remaining plans or packages by using DSNTIAUL to select the plans or packages from the DB2 catalog and to create the REBIND subcommands. You can then submit the subcommands through the DSN command processor, as usual.

You might first need to edit the output from DSNTIAUL so that DSN can accept it as input. The CLIST DSNTEDIT can perform much of that task for you.

This section contains the following topics:

- “Generating lists of REBIND commands”
- “Sample SELECT statements for generating REBIND commands” on page 966
- “Sample JCL for running lists of REBIND commands” on page 968

Sample SELECT statements for generating REBIND commands

You can select specific plans or packages to be rebound and concatenate the REBIND subcommand syntax around the plan or package names. You can also convert a varying-length string to a fixed-length string, and append additional blanks to the REBIND PLAN and REBIND PACKAGE subcommands, so that the DSN command processor can accept the record length as valid input.

Building REBIND subcommands: The examples that follow illustrate the following techniques:

- Using SELECT to select specific packages or plans to be rebound
- Using the CONCAT operator to concatenate the REBIND subcommand syntax around the plan or package names
- Using the SUBSTR function to convert a varying-length string to a fixed-length string
- Appending additional blanks to the REBIND PLAN and REBIND PACKAGE subcommands, so that the DSN command processor can accept the record length as valid input

If the SELECT statement returns rows, then DSNTIAUL generates REBIND subcommands for the plans or packages identified in the returned rows. Put those subcommands in a sequential data set, where you can then edit them.

For REBIND PACKAGE subcommands, delete any extraneous blanks in the package name, using either TSO edit commands or the DB2 CLIST DSNTEDIT.

For both REBIND PLAN and REBIND PACKAGE subcommands, add the DSN command that the statement needs as the first line in the sequential data set, and add END as the last line, using TSO edit commands. When you have edited the sequential data set, you can run it to rebound the selected plans or packages.

If the SELECT statement returns no qualifying rows, then DSNTIAUL does not generate REBIND subcommands.

The examples in this topic generate REBIND subcommands that work in DB2 for z/OS Version 10. You might need to modify the examples for prior releases of DB2 that do not allow all of the same syntax.

Example: REBIND all plans without terminating because of unavailable resources.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT')
',1,45)
FROM SYSIBM.SYSPLAN;
```

Example: REBIND all versions of all packages without terminating because of unavailable resources.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'. '
CONCAT NAME CONCAT'.*)
',1,55)
FROM SYSIBM.SYSPACKAGE;
```

Example: REBIND all plans bound before a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT')
',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE <= 'yymmdd' OR
(BINDDATE <= 'yymmdd' AND
BINDTIME <= 'hhmmssst');
```


where *yymmdd* represents the date portion and *hhmmssst* represents the time portion of the timestamp string.

If the date specified is after 2000, you need to include another condition that includes plans that were bound before year 2000:

```
WHERE
    BINDDATE >= '830101' OR
    BINDDATE <= 'yymmdd' OR
    (BINDDATE <= 'yymmdd' AND
     BINDTIME <= 'hhmmssst');
```

Example: REBIND all versions of all packages bound before a given date and time.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'. '
    CONCAT NAME CONCAT'.(*)'          ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME <= 'timestamp';
```

where *timestamp* is an ISO timestamp string.

Example: REBIND all plans bound since a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
    CONCAT')'          ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE >= 'yymmdd' AND
     BINDTIME >= 'hhmmssst';
```

where *yymmdd* represents the date portion and *hhmmssst* represents the time portion of the timestamp string.

Example: REBIND all versions of all packages bound since a given date and time.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID
    CONCAT'. 'CONCAT NAME
    CONCAT'.(*)'          ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp';
```

where *timestamp* is an ISO timestamp string.

Example: REBIND all plans bound within a given date and time range.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
    CONCAT')'          ',1,45)
FROM SYSIBM.SYSPLAN
WHERE
    (BINDDATE >= 'yymmdd' AND
     BINDTIME >= 'hhmmssst') AND
    BINDDATE <= 'yymmdd' AND
    BINDTIME <= 'hhmmssst');
```

where *yymmdd* represents the date portion and *hhmmssst* represents the time portion of the timestamp string.

Example: REBIND all versions of all packages bound within a given date and time range.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'. '
    CONCAT NAME CONCAT'.(*)'          ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp1' AND
     BINDTIME <= 'timestamp2';
```

where *timestamp1* and *timestamp2* are ISO timestamp strings.

Example: REBIND all invalid versions of all packages.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'. '
CONCAT NAME CONCAT'.(*)          ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE VALID = 'N';
```

Example: REBIND all plans bound with ISOLATION level of cursor stability.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT')          ',1,45)
FROM SYSIBM.SYSPLAN
WHERE ISOLATION = 'S';
```

Example: REBIND all versions of all packages that allow CPU and/or I/O parallelism.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'. '
CONCAT NAME CONCAT'.(*)          ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE DEGREE='ANY';
```

Sample JCL for running lists of REBIND commands

You can use JCL to rebind all versions of all packages that are bound within a specified date and time period.

You specify the date and time period for which you want packages to be rebound in a WHERE clause of the SELECT statement that contains the REBIND command. In The following example, the WHERE clause looks like the following clause:

```
WHERE BINDTIME >= 'YYYY-MM-DD-hh.mm.ss' AND
      BINDTIME <= 'YYYY-MM-DD-hh.mm.ss'
```

The date and time period has the following format:

YYYY The four-digit year. For example: 2008.

MM The two-digit month, which can be a value between 01 and 12.

DD The two-digit day, which can be a value between 01 and 31.

hh The two-digit hour, which can be a value between 01 and 24.

mm The two-digit minute, which can be a value between 00 and 59.

ss The two-digit second, which can be a value between 00 and 59.

```
//REBINDS JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//          REGION=1024K
//*****
//SETUP EXEC PGM=IKJEFT01
//SYSTSPT DD SYSOUT=*
//SYSTSIN DD *
      DSN SYSTEM(DSN)
      RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBA1) PARM('SQL') -
          LIB('DSNA10.RUNLIB.LOAD')
      END
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//          UNIT=SYSDA,DISP=SHR
//*****
//*
//* GENER= '<SUBCOMMANDS TO REBIND ALL PACKAGES BOUND IN YYYY
//*
//*****
//SYSIN DD *
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'. '
CONCAT NAME CONCAT'.(*)          ',1,55)
```

```

FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'YYYY-MM-DD-hh.mm.ss' AND
      BINDTIME <= 'YYYY-MM-DD-hh.mm.ss';
/*
//*****
//*
//* STRIP THE BLANKS OUT OF THE REBIND SUBCOMMANDS
//*
//*****
//STRIP EXEC PGM=IKJEFT01
//SYSPROC DD DSN=SYSADM.DSNCLIST,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SYSTSIN DD *
      DSNTEEDIT SYSADM.SYSTSIN.DATA
//SYSIN DD DUMMY
/*
//*****
//*
//* PUT IN THE DSN COMMAND STATEMENTS
//*
//*****
//EDIT EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
      EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
      TOP
      INSERT DSN SYSTEM(DSN)
      BOTTOM
      INSERT END
      TOP
      LIST * 99999
      END SAVE
/*
//*****
//*
//* EXECUTE THE REBIND PACKAGE SUBCOMMANDS THROUGH DSN
//*
//*****
//LOCAL EXEC PGM=IKJEFT01
//DBRMLIB DD DSN=DSNA10.DBRMLIB.DATA,
//      DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD DSN=SYSADM.SYSTSIN.DATA,
//      UNIT=SYSDA,DISP=SHR
/*

```

The following example shows some sample JCL for rebinding all plans bound without specifying the DEGREE keyword on BIND with DEGREE(ANY).

```

//REBINDS JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//      REGION=1024K
//*****
//SETUP EXEC TSOBATCH
//SYSPRINT DD SYSOUT=*
//SYSPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//      UNIT=SYSDA,DISP=SHR
//*****
//*
//* REBIND ALL PLANS THAT WERE BOUND WITHOUT SPECIFYING THE DEGREE
//* KEYWORD ON BIND WITH DEGREE(ANY)
//*
//*****

```

```

//SYSTSIN DD *
DSN S(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBA1) PARM('SQL')
END
//SYSIN DD *
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT') DEGREE(ANY)           ',1,45)
FROM SYSIBM.SYSPLAN
WHERE DEGREE = ' ';
/*
//*****
//*
//* PUT IN THE DSN COMMAND STATEMENTS
//*
//*****
//EDIT EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
TOP
INSERT DSN S(DSN)
BOTTOM
INSERT END
TOP
LIST * 99999
END SAVE
/*
//*****
//*
//* EXECUTE THE REBIND SUBCOMMANDS THROUGH DSN
//*
//*****
//REBIND EXEC PGM=IKJEFT01
//STEPLIB DD DSN=SYSADM.TESTLIB,DISP=SHR
// DD DSN=DSNA10.SDSNLOAD,DISP=SHR
//DBRMLIB DD DSN=SYSADM.DBRMLIB.DATA,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SYSTSIN DD DSN=SYSADM.SYSTSIN.DATA,DISP=SHR
//SYSIN DD DUMMY
/*

```

Automatic rebinding

Automatic rebind might occur if an authorized user invokes a package under some situations. These situations include when the attributes of the data on which the package depends change, or if the environment in which the package executes changes. Whether the automatic rebind occurs depends on the value of the ABIND subsystem parameter.

In general, the option values that are used for an automatic rebind are the values that were used during the most recent bind process. Exceptions are:

- If an option is no longer supported, the automatic rebind option process substitutes a supported option.
- The automatic rebind value for APCOMPARE and APREUSE is NONE.

If a package has previous or original copies as a result of rebinding with the PLANMGMT(BASIC) or PLANMGMT(EXTENDED) options or having the PLANMGMT subsystem parameter set to BASIC or EXTENDED, those copies are not affected by automatic rebind. Automatic rebind replaces only the current copy.

In most cases, DB2 marks a package that needs to be automatically rebound as invalid. A few common situations in which DB2 marks a package as invalid are:

- When a package is dropped
- When a plan depends on the execute privilege of a package that is dropped
- When a table, index, or view on which the package depends is dropped
- When the authorization of the owner to access a table, index, or view on which the package depends is revoked
- When the authorization to execute a stored procedure is revoked from a package owner, and the package uses the CALL *procedure-name* form of the CALL statement to call the stored procedure
- When a table on which the package depends is altered to add a TIME, TIMESTAMP, or DATE column
- When a table is altered to add a self-referencing constraint or a constraint with a delete rule of SET NULL or CASCADE
- When the limit key value of a partitioned index on which the package depends is altered
- When the definition of an index on which the package depends is altered from NOT PADDED to PADDED
- When the definition of an index on which the package depends is altered from PADDED to NOT PADDED
- When the AUDIT attribute of a table on which the package depends is altered
- When the length attribute of a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY column in a table on which the package depends is altered
- When the data type, precision, or scale of a column in a table on which the package depends is altered
- When a package depends on a view that DB2 cannot regenerate after a column in the underlying table is altered
- When a created temporary table on which the package depends is altered to add a column
- When a user-defined function on which the package depends is altered
- When a column is renamed in a table on which a package is dependent
- When a plan or package depends on a procedure that is regenerated
- When a pre-V10 plan has DBRMs bound directly to it. If subsystem parameter DISALLOW_DEFAULT_COLLID is set to NO, automatic rebind binds the DBRMs into packages, and binds the packages into plans with the collection ID DSN_DEFAULT_COLLID_ *plan-name*.

Whether a package is valid is recorded in column VALID of catalog tables SYSPLAN and SYSPACKAGE.

In the following cases, DB2 automatically rebinds a package that has not been marked as invalid if the ABIND subsystem parameter is set to YES (the default):

- A package that is bound on a release of DB2 that is more recent than the release in which it is being run. This situation can happen in a data sharing environment or after a DB2 subsystem has fallen back to a previous release of DB2.
- In Version 10, plans and packages that were bound in releases earlier than Version 6 Release 1 are not supported. Such packages are automatically rebound when they are run in the current DB2 release.

- A package that has a location dependency and runs at a location other than the one at which it was bound. This situation can happen when members of a data sharing group are defined with location names, and a package runs on a different member from the one on which it was bound.

In the following cases, DB2 automatically rebinds a package that has not been marked as invalid if the ABIND subsystem parameter is set to COEXIST:

- The subsystem on which the package runs is in a data sharing group.
- The package was previously bound on the current DB2 release and is now running on the previous DB2 release.

If the ABIND subsystem parameter is set to NO and you attempt to execute a package that requires a rebind, but cannot be automatically rebound, DB2 returns an error.

DB2 marks a package as *inoperative* if an automatic rebind fails. Whether a package is operative is recorded in column OPERATIVE of SYSPLAN and SYSPACKAGE.

Whether EXPLAIN runs during automatic rebind depends on the value of the field EXPLAIN PROCESSING on installation panel DSNTIPO, and on whether you specified EXPLAIN(YES). Automatic rebind fails for all EXPLAIN errors except "PLAN_TABLE not found."

The SQLCA is not available during automatic rebind. Therefore, if you encounter lock contention during an automatic rebind, DSNT501I messages cannot accompany any DSNT376I messages that you receive. To see the matching DSNT501I messages, you must issue the subcommand REBIND PLAN or REBIND PACKAGE.

If an automatic bind occurs while running in ACCESS(MAINT) mode the automatic bind is run under the authorization id of SYSOPR. If SYSOPR is not defined as an installation SYSOPR the automatic bind fails.

Related tasks:

 Disallowing all automatic rebinds (DB2 Installation and Migration)

Related reference:

 AUTO BIND field (ABIND subsystem parameter) (DB2 Installation and Migration)

Specifying the rules that apply to SQL behavior at run time

You can specify whether DB2 rules or SQL standard rules apply to SQL behavior at run time.

Not only does SQLRULES specify the rules under which a type 2 CONNECT statement executes, but it also sets the initial value of the special register CURRENT RULES when the database server is the local DB2 system. When the server is not the local DB2 system, the initial value of CURRENT RULES is DB2. After binding a plan, you can change the value in CURRENT RULES in an application program by using the statement SET CURRENT RULES.

CURRENT RULES determines the SQL rules, DB2 or SQL standard, that apply to SQL behavior at run time. For example, the value in CURRENT RULES affects the behavior of defining check constraints by issuing the ALTER TABLE statement on a populated table:

- If **CURRENT RULES** has a value of **STD** and no existing rows in the table violate the check constraint, DB2 adds the constraint to the table definition. Otherwise, an error occurs and DB2 does not add the check constraint to the table definition.

If the table contains data and is already in a check pending status, the ALTER TABLE statement fails.

- If **CURRENT RULES** has a value of **DB2**, DB2 adds the constraint to the table definition, defers the enforcing of the check constraints, and places the table space or partition in CHECK-pending status.

You can use the statement SET CURRENT RULES to control the action that the statement ALTER TABLE takes. Assuming that the value of CURRENT RULES is initially STD, the following SQL statements change the SQL rules to DB2, add a check constraint, defer validation of that constraint, place the table in CHECK-pending status, and restore the rules to STD.

```
EXEC SQL
  SET CURRENT RULES = 'DB2';
EXEC SQL
  ALTER TABLE DSN8A10.EMP
  ADD CONSTRAINT C1 CHECK (BONUS <= 1000.0);
EXEC SQL
  SET CURRENT RULES = 'STD';
```

See “Check constraints” on page 469 for information about check constraints.

You can also use CURRENT RULES in host variable assignments. For example, if you want to store the value of the CURRENT RULES special register at a particular point in time, you can use assign the value to a host variable, as in the following statement:

```
SET :XRULE = CURRENT RULES;
```

You can also use CURRENT RULES as the argument of a search-condition. For example, the following statement retrieves rows where the COL1 column contains the same value as the CURRENT RULES special register.

```
SELECT * FROM SAMPTBL WHERE COL1 = CURRENT RULES;
```

DB2 program preparation overview

Before you can run an application program on DB2 for z/OS, you need to prepare it. To prepare the program, create a load module, possibly one or more packages, and an application plan.

If your application program includes SQL statements, you need to process those SQL statements by using either the DB2 precompiler or the DB2 coprocessor that is provided with a compiler. Both the precompiler and the coprocessor perform the following actions:

- Replaces the SQL statements in your source programs with calls to DB2 language interface modules
- Creates a database request module (DBRM), which communicates your SQL requests to DB2 during the bind process

The following figure illustrates the program preparation process when you use the DB2 precompiler. After you process SQL statements in your source program by using the DB2 precompiler, you create a load module, possibly one or more packages, and an application plan. Creating a load module involves compiling the

modified source code that is produced by the precompiler into an object program, and link-editing the object program to create a load module. Creating a package or an application plan, a process unique to DB2, involves binding one or more DBRMs, which are created by the DB2 precompiler, using the BIND PACKAGE command.

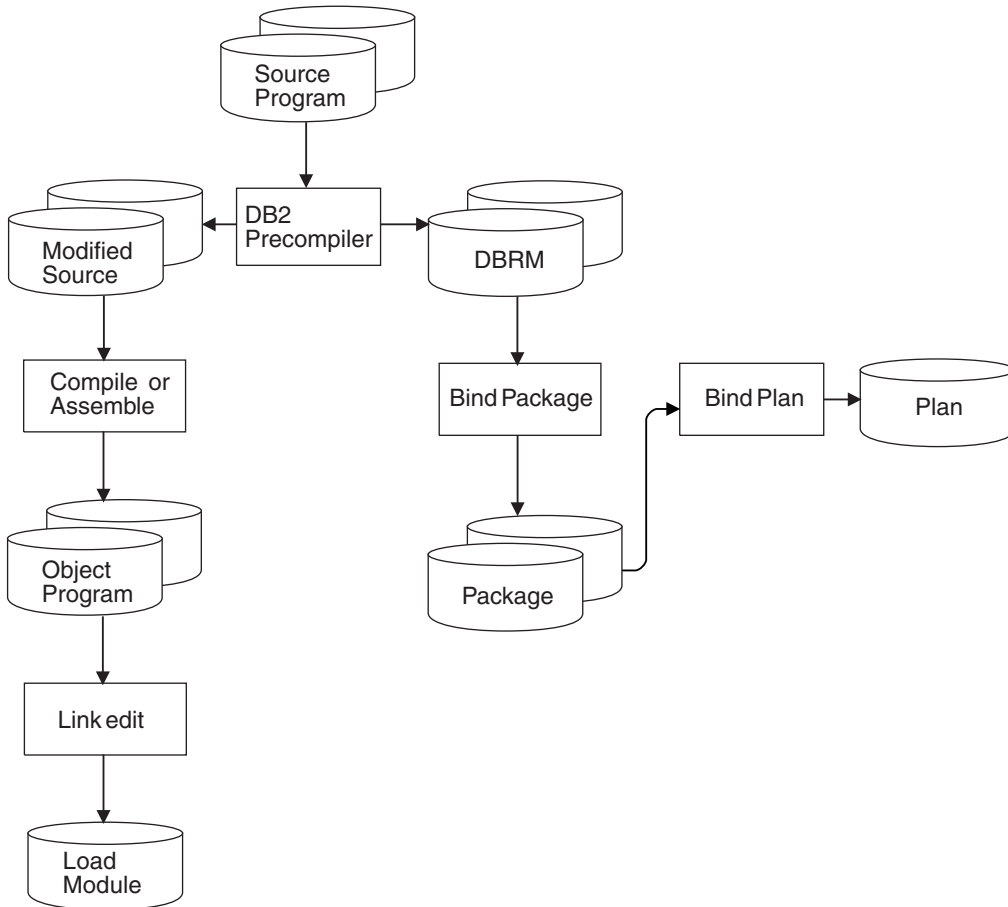


Figure 51. Program preparation with the DB2 precompiler

The following figure illustrates the program preparation process when you use the DB2 coprocessor. The process is similar to the process for the DB2 precompiler, except that the DB2 coprocessor does not create modified source for your application program.

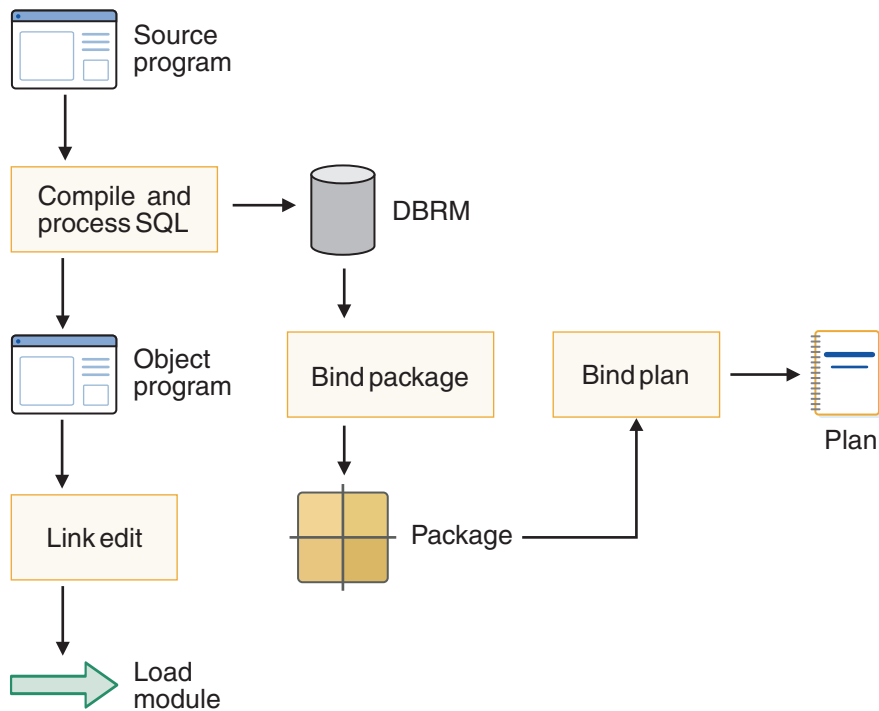


Figure 52. Program preparation with the DB2 coprocessor

Input and output data sets for DL/I batch jobs

DL/I batch jobs require an input data set with DD name DDITV02 and an output data set with DD name DDOTV02.

DB2 DL/I batch input:

Before you can run a DL/I batch job, you need to provide values for a number of input parameters. The input parameters are positional and delimited by commas.

You can specify values for the following parameters using a DDITV02 data set or a subsystem member:

SSN,LIT,ESMT,RTT,REQ,CRC

You can specify values for the following parameters **only** in a DDITV02 data set:

CONNECTION_NAME,PLAN,PROG

If you use the DDITV02 data set and specify a subsystem member, the values in the DDITV02 DD statement override the values in the specified subsystem member. If you provide neither, DB2 abnormally terminates the application program with system abend code X'04E' and a unique reason code in register 15.

DDITV02 is the DD name for a data set that has DCB options of LRECL=80 and RECFM=F or FB.

A subsystem member is a member in the IMS procedure library. Its name is derived by concatenating the value of the SSM parameter to the value of the

IMSID parameter. You specify the SSM parameter and the IMSID parameter when you invoke the DLIBATCH procedure, which starts the DL/I batch processing environment.

The meanings of the input parameters are:

Field	Content
-------	---------

SSN	Specifies the name of the DB2 subsystem. This value is required. You must specify a name in order to make a connection to DB2.
------------	--

The SSN value can be from one to four characters long.

If the value in the SSN parameter is the name of an active subsystem in the data sharing group, the application attaches to that subsystem. If the SSN parameter value is not the name of an active subsystem, but the value is a group attachment name, the application attaches to an active DB2 subsystem in the data sharing group.

LIT	Specifies a language interface token. DB2 requires a language interface token to route SQL statements when operating in the online IMS environment. Because a batch application program can connect to only one DB2 system, DB2 does not use the LIT value.
------------	---

The LIT value can be from zero to four characters long.

Recommendation: Specify the LIT value as SYS1.

You can omit the LIT value by entering SSN,,ESMT.

ESMT	Specifies the name of the DB2 initialization module, DSNMIN10. This value is required.
-------------	--

The ESMT value must be eight characters long.

RTT	Specifies the resource translation table. This value is optional.
------------	---

The RTT can be from zero to eight characters long.

REO	Specifies the region error option. This option determines what to do if DB2 is not operational or the plan is not available. The three options are:
------------	---

- *R*, the default, results in returning an SQL return code to the application program. The most common SQLCODE issued in this case is -923 (SQLSTATE '57015').
- *Q* results in an abend in the batch environment; however, in the online environment, this value places the input message in the queue again.
- *A* results in an abend in both the batch environment and the online environment.

If the application program uses the XRST call, and if coordinated recovery is required on the XRST call, REO is ignored. In that case, the application program terminates abnormally if DB2 is not operational.

The REO value can be from zero to one character long.

CRC	Specifies the command recognition character. Because DB2 commands are not supported in the DL/I batch environment, the command recognition character is not used at this time.
------------	--

The CRC value can be from zero to one character long.

CONNECTION_NAME

Represents the name of the job step that coordinates DB2 activities. This value is optional. If you do not specify this option, the connection name defaults are:

Type of application

Default connection name

Batch job

Job name

Started task

Started task name

TSO user

TSO authorization ID

If a batch update job fails, you must use a separate job to restart the batch job. The connection name used in the restart job must be the same as the name that is used in the batch job that failed. Alternatively, if the default connection name is used, the restart job must have the same job name as the batch update job that failed.

DB2 requires unique connection names. If two applications try to connect with the same connection name, the second application program fails to connect to DB2.

The CONNECTION_NAME value can be from one to eight characters long.

PLAN Specifies the DB2 plan name. This value is optional. If you do not specify the plan name, the application program module name is checked against the optional resource translation table. If the resource translation table has a match, the translated name is used as the DB2 plan name. If no match exists in the resource translation table, the application program module name is used as the plan name.

The PLAN value can be from zero to eight characters long.

PROG Specifies the application program name. This value is required. It identifies the application program that is to be loaded and to receive control.

The PROG value can be from one to eight characters long.

Example: An example of the fields in the record is shown below:

```
DSN,SYS1,DSNMIN10,,R,-,BATCH001,DB2PLAN,PROGA
```

DB2 DL/I batch output:

In an online IMS environment, DB2 sends unsolicited status messages to the master terminal operator (MTO) and records on indoubt processing and diagnostic information to the IMS log. In a batch environment, DB2 sends this information to the output data set that is specified in the DDOTV02 DD statement. Ensure that the output data set has DCB options of RECFM=V or VB, LRECL=4092, and BLKSIZE of at least LRECL + 4. If the DD statement is missing, DB2 issues the message IEC130I and continues processing without any output.

You might want to save and print the data set, as the information is useful for diagnostic purposes. You can use the IMS module, DFSERA10, to print the variable-length data set records in both hexadecimal and character format.

Related concepts:

☛ Submitting work to be processed (DB2 Data Sharing Planning and Administration)

DB2-supplied JCL procedures for preparing an application

You can precompile and prepare an application program using a DB2-supplied JCL procedure.

DB2 has a unique JCL procedure for each supported language, with appropriate defaults for starting the DB2 precompiler and host language compiler or assembler. The procedures are in *prefix.SDSNSAMP* member *DSNTIJMV*, which installs the procedures.

Table 158. Procedures for precompiling programs

Language	Procedure	Invocation included in...
High-level assembler	DSNHASM	DSNTEJ2A
C	DSNHHC	DSNTEJ2D
C++	DSNHCPPDSNHCPP2 ²	DSNTEJ2EN/A
Enterprise COBOL	DSNHICOB	DSNTEJ2C ¹
Fortran	DSNHFOR	DSNTEJ2F
PL/I	DSNHPLI	DSNTEJ2P
SQL	DSNHSQL	DSNTEJ63

Notes:

1. You must customize these programs to invoke the procedures that are listed in this table.
2. This procedure demonstrates how you can prepare an object-oriented program that consists of two data sets or members, both of which contain SQL.

If you use the PL/I macro processor, you must not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the PARM.PLI= parameter of the EXEC statement in the DSNHPLI procedure.

JCL to include the appropriate interface code when using the DB2-supplied JCL procedures

To include the proper interface code when you submit the JCL procedures, use an INCLUDE SYSLIB statement in your link-edit JCL. The statement should specify the correct language interface module for the environment.

TSO, batch

```
//LKED.SYSIN DD *  
  INCLUDE SYSLIB(member)  
/*
```

member must be DSNELI or DSNULI, except for FORTRAN, in which case *member* must be DSNHFT.

IMS

```
//LKED.SYSIN DD *  
  INCLUDE SYSLIB(DFSLI000)  
  ENTRY (specification)  
/*
```

DFSLI000 is the module for DL/I batch attach.

ENTRY *specification* varies depending on the host language. Include one of the following:

DLITCBL, for COBOL applications

PLICALLA, for PL/I applications

The program name, for assembler language applications.

Recommendation: For COBOL applications, specify the PSB linkage directly on the PROCEDURE DIVISION statement instead of on a DLITCBL entry point. When you specify the PSB linkage directly on the PROCEDURE DIVISION statement, you can either omit the ENTRY specification or specify the application program name instead of the DLITCBL entry point.

CICS

```
//LKED.SYSIN DD *  
    INCLUDE SYSLIB(member)  
/*
```

member must be DSNCLI or DSNULI.

Related concepts:

“Universal language interface” on page 153

Related tasks:

“Making the CAF language interface (DSNALI) available” on page 80

“Compiling and link-editing an application” on page 942

Tailoring DB2-supplied JCL procedures for preparing CICS programs

Instead of using the DB2 Program Preparation panels to prepare your CICS program, you can tailor CICS-supplied JCL procedures to do that. To tailor a CICS procedure, you need to add some steps and change some DD statements.

Make changes as needed to perform the following actions:

- Process the program with the DB2 precompiler.
- Bind the application plan. You can do this any time after you precompile the program. You can bind the program either online by the DB2I panels or as a batch step in this or another z/OS job.
- Include a DD statement in the linkage editor step to access the DB2 load library.
- Be sure the linkage editor control statements contain an INCLUDE statement for the DB2 language interface module.

The following example illustrates the necessary changes. This example assumes the use of a COBOL program. For any other programming language, change the CICS procedure name and the DB2 precompiler options.

```
    //TESTC01 JOB  
    /*  
    /******  
    /*          DB2 PRECOMPILE THE COBOL PROGRAM  
    /******  
(1) //PC      EXEC PGM=DSNHPC,  
(1) //        PARM='HOST(COB2),XREF,SOURCE,FLAG(1),APOST'  
(1) //STEPLIB DD DISP=SHR,DSN=prefix.SDSNEXIT  
(1) //        DD DISP=SHR,DSN=prefix.SDSNLOAD  
(1) //DBRMLIB DD DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)  
(1) //SYSCIN  DD DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
```

```

(1) //          SPACE=(800,(500,500))
(1) //SYSLIB DD DISP=SHR,DSN=USER.SRCLIB.DATA
(1) //SYSPRINT DD SYSOUT=*
(1) //SYSTEM DD SYSOUT=*
(1) //SYSUDUMP DD SYSOUT=*
(1) //SYSUT1 DD SPACE=(800,(500,500),,ROUND),UNIT=SYSDA
(1) //SYSUT2 DD SPACE=(800,(500,500),,ROUND),UNIT=SYSDA
(1) //SYSIN DD DISP=SHR,DSN=USER.SRCLIB.DATA(TESTC01)
(1) //*

//*****
//*** BIND THIS PROGRAM.
//*****

(2) //BIND EXEC PGM=IKJEFT01,
(2) // COND=(4,LT,PC)
(2) //STEPLIB DD DISP=SHR,DSN=prefix.SDSNEXIT
(2) // DD DISP=SHR,DSN=prefix.SDSNLOAD
(2) //DBRMLIB DD DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
(2) //SYSPRINT DD SYSOUT=*
(2) //SYSSTPRT DD SYSOUT=*
(2) //SYSUDUMP DD SYSOUT=*
(2) //SYSYSIN DD *
(2) DSN S(DSN)
(2) BIND PLAN(TESTC01) MEMBER(TESTC01) ACTION(REP) RETAIN ISOLATION(CS)
(2) END

//*****
//* COMPILE THE COBOL PROGRAM
//*****

(3) //CICS EXEC DFHEITVL
(4) //TRN.SYSIN DD DSN=&&DSNHOUT,DISP=(OLD,DELETE)
(5) //LKED.SYSLMOD DD DSN=USER.RUNLIB.LOAD
(6) //LKED.CICSLOAD DD DISP=SHR,DSN=prefix.SDFHLOAD
//LKED.SYSIN DD *
(7) INCLUDE CICSLOAD(DSNCLI)
NAME TESTC01(R)
//*****

```

The procedure accounts for these steps:

- Step 1.** Precompile the program. The output of the DB2 precompiler becomes the input to the CICS command language translator.
- Step 2.** Bind the application plan.
- Step 3.** Call the CICS procedure to translate, compile, and link-edit a COBOL program. This procedure has several options that you need to consider.
- Step 4.** Reflect an application load library in the data set name of the SYSLMOD DD statement. You must include the name of this load library in the DFHRPL DD statement of the CICS run time JCL.
- Step 5.** Name the CICS load library that contains the module DSNCLI.
- Step 6.** Direct the linkage editor to include the CICS-DB2 language interface module (DSNCLI). In this example, the order of the various control sections (CSECTs) is of no concern because the structure of the procedure automatically satisfies any order requirements.

For more information about the procedure DFHEITVL, other CICS procedures, or CICS requirements for application programs, please see the appropriate CICS manual.

If you are preparing a particularly large or complex application, you can use another preparation method. For example, if your program requires four of your own link-edit include libraries, you cannot prepare the program with DB2I, because DB2I limits the number of include libraries to three, plus language, IMS or CICS, and DB2 libraries. Therefore, you would need another preparation method.

Be careful to use the correct language interface.

DB2I primary option menu

The DB2I Primary Option menu is the starting point for all DB2I tasks.

Figure 53 shows an example of the DB2I Primary Option Menu. From this point, you can access all of the DB2I panels without passing through panels that you do not need. For example, to bind a program, enter the number that corresponds to BIND/REBIND/FREE to reach the BIND PLAN panel without seeing the ones previous to it.

```
DSNEPRI          DB2I PRIMARY OPTION MENU          SSID: DSN
COMMAND ==>> 3_

Select one of the following DB2 functions and press ENTER.

 1 SPUFI          (Process SQL statements)
 2 DCLGEN         (Generate SQL and source language declarations)
 3 PROGRAM PREPARATION (Prepare a DB2 application program to run)
 4 PRECOMPILE    (Invoke DB2 precompiler)
 5 BIND/REBIND/FREE (BIND, REBIND, or FREE plans or packages)
 6 RUN           (RUN an SQL program)
 7 DB2 COMMANDS  (Issue DB2 commands)
 8 UTILITIES    (Invoke DB2 utilities)
 D DB2I DEFAULTS (Set global parameters)
 X EXIT         (Leave DB2I)
```

Figure 53. Initiating program preparation through DB2I. Specify Program Preparation on the DB2I Primary Option Menu.

The following descriptions explain the functions on the DB2I Primary Option Menu.

1 SPUFI

Lets you develop and execute one or more SQL statements interactively. For further information, see “Executing SQL by using SPUFI” on page 1035.

2 DCLGEN

Lets you generate C, COBOL, or PL/I data declarations of tables. For further information, see “DCLGEN (declarations generator)” on page 161.

3 PROGRAM PREPARATION

Lets you prepare and run an application program to run. For more information, see “DB2 Program Preparation panel” on page 983.

4 PRECOMPILE

Lets you convert embedded SQL statements into statements that your host language can process. For further information, see “Precompile panel” on page 990.

5 BIND/REBIND/FREE

Lets you bind, rebind, or free a package or application plan. For more information, see “Bind/Rebind/Free Selection panel” on page 1008.

6 RUN

Lets you run an application program in a TSO or batch environment. For more information, see “DB2I Run panel” on page 1018.

7 DB2 COMMANDS

Lets you issue DB2 commands.

8 UTILITIES

Lets you call DB2 utility programs.

D DB2I DEFAULTS

Lets you set DB2I defaults. See “DB2I Defaults Panel 1” on page 987.

X EXIT

Lets you exit DB2I.

DB2I panels that are used for program preparation

DB2I contains a set of panels that let you prepare an application for execution.

The following table describes each of the panels that you need to use to prepare an application.

Table 159. DB2I panels used for program preparation

Panel name	Panel description
DB2 Program Preparation “DB2 Program Preparation panel” on page 983	Lets you choose specific program preparation functions to perform. For the functions that you choose, you can also display the associated panels to specify options for performing those functions. This panel also lets you change the DB2I default values and perform other precompile and prelink functions.
DB2I Defaults Panel 1 “DB2I Defaults Panel 1” on page 987	Lets you change many of the system defaults that are set at DB2 installation time.
DB2I Defaults Panel 2 “DB2I Defaults Panel 2” on page 989	Lets you change your default job statement and set additional COBOL options.
Precompile “Precompile panel” on page 990	Lets you specify values for precompile functions. You can reach this panel directly from the DB2I Primary Option Menu or from the DB2 Program Preparation panel. If you reach this panel from the Program Preparation panel, many of the fields contain values from the Primary and Precompile panels.
Bind Package “Bind Package panel” on page 993	Lets you change many options when you bind a package. You can reach this panel directly from the DB2I Primary Option Menu or from the DB2 Program Preparation panel. If you reach this panel from the DB2 Program Preparation panel, many of the fields contain values from the Primary and Precompile panels.
Bind Plan “Bind Plan panel” on page 996	Lets you change options when you bind an application plan. You can reach this panel directly from the DB2I Primary Option Menu or as a part of the program preparation process. This panel also follows the Bind Package panels.
Defaults for Bind or Rebind Package or Plan panels “Defaults for Bind Package and Defaults for Rebind Package panels” on page 998	Let you change the defaults for BIND or REBIND PACKAGE or PLAN.
System Connection Types panel “System Connection Types panel” on page 1003	Lets you specify a system connection type. This panel displays if you choose to enable or disable connections on the Bind or Rebind Package or Plan panels.

Table 159. DB2I panels used for program preparation (continued)

Panel name	Panel description
Panels for entering lists of values“Panels for entering lists of values” on page 1004	Let you enter or modify an unlimited number of values. A list panel looks similar to an ISPF edit session and lets you scroll and use a limited set of commands.
Program Prep: Compile, Prelink, Link, and Run“Program Preparation: Compile, Link, and Run panel” on page 1005	Lets you perform the last two steps in the program preparation process (compile and link-edit). This panel also lets you do the PL/I MACRO PHASE for programs that require this option. For TSO programs, the panel also lets you run programs.

DB2 Program Preparation panel

The DB2 Program Preparation panel lets you choose which specific program preparation function to perform.

For the functions you choose, you can also choose to display the associated panels to specify options for performing those functions. Some of the functions you can select are:

Precompile

The panel for this function lets you control the DB2 precompiler.

Bind a package

The panel for this function lets you bind your program's DBRM to a package and change your defaults for binding the packages.

Bind a plan

The panel for this function lets you create your program's application plan and change your defaults for binding the plans.

Compile, link, and run

The panel for these functions let you control the compiler or assembler and the linkage editor.

TSO and batch: For TSO programs, you can use the program preparation programs to control the host language run time processor and the program itself.

The Program Preparation panel also lets you change the DB2I default values, and perform other precompile and prelink functions.

On the DB2 Program Preparation panel, shown in the following figure, enter the name of the source program data set (this example uses SAMPLEPG.COBOL) and specify the other options you want to include. When finished, press ENTER to view the next panel.

```

DSNEPP01          DB2 PROGRAM PREPARATION          SSID: DSN
COMMAND ==>_

Enter the following:
 1 INPUT DATA SET NAME .... ==> SAMPLEPG.COBOL
 2 DATA SET NAME QUALIFIER ==> TEMP      (For building data set names)
 3 PREPARATION ENVIRONMENT ==> FOREGROUND (FOREGROUND, BACKGROUND, EDITJCL)
 4 RUN TIME ENVIRONMENT ... ==> TSO      (TSO, CAF, CICS, IMS, RRSAF)
 5 OTHER DSNH OPTIONS ..... ==>

Select functions:          Display panel?          (Optional DSNH keywords)
                        Perform function?
 6 CHANGE DEFAULTS ..... ==> Y (Y/N)
 7 PL/I MACRO PHASE ..... ==> N (Y/N)          ==> N (Y/N)
 8 PRECOMPILE ..... ==> Y (Y/N)              ==> Y (Y/N)
 9 CICS COMMAND TRANSLATION
10 BIND PACKAGE ..... ==> Y (Y/N)            ==> Y (Y/N)
11 BIND PLAN..... ==> Y (Y/N)                ==> Y (Y/N)
12 COMPILE OR ASSEMBLE ... ==> Y (Y/N)        ==> Y (Y/N)
13 PRELINK..... ==> N (Y/N)                  ==> N (Y/N)
14 LINK..... ==> N (Y/N)                      ==> Y (Y/N)
15 RUN..... ==> N (Y/N)                       ==> Y (Y/N)

```

Figure 54. The DB2 Program Preparation panel. Enter the source program data set name and other options.

The following explains the functions on the DB2 Program Preparation panel and how to complete the necessary fields in order to start program preparation.

1 INPUT DATA SET NAME

Lets you specify the input data set name. The input data set name can be a PDS or a sequential data set, and can also include a member name. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) qualifies the data set name.

The input data set name you specify is used to precompile, bind, link-edit, and run the program.

2 DATA SET NAME QUALIFIER

Lets you qualify temporary data set names involved in the program preparation process. Use any character string from 1 to 8 characters that conforms to normal TSO naming conventions. (The default is TEMP.)

For programs that you prepare in the background or that use EDITJCL for the PREPARATION ENVIRONMENT option, DB2 creates a data set named *tsoprefix.qualifier.CNTL* to contain the program preparation JCL. The name *tsoprefix* represents the prefix TSO assigns, and *qualifier* represents the value you enter in the DATA SET NAME QUALIFIER field. If a data set with this name already exists, DB2 deletes it.

3 PREPARATION ENVIRONMENT

Lets you specify whether program preparation occurs in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job. Use:

FOREGROUND to use the values you specify on the Program Preparation panel and to run immediately.

BACKGROUND to create and submit a file containing a DSNH CLIST that runs immediately using the JOB control statement from either the DB2I Defaults panel or your site's SUBMIT exit. The file is saved.

EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

4 RUN TIME ENVIRONMENT

Lets you specify the environment (TSO, CAF, CICS, IMS, RRSAF) in which your program runs.

All programs are prepared under TSO, but can run in any of the environments. If you specify CICS, IMS, or RRSAF, then you must set the RUN field to NO because you cannot run such programs from the Program Preparation panel. If you set the RUN field to YES, you can specify only TSO or CAF.

(Batch programs also run under the TSO Terminal Monitor Program. You therefore need to specify TSO in this field for batch programs.)

5 OTHER DSNH OPTIONS

Lets you specify a list of DSNH options that affect the program preparation process, and that override options specified on other panels. If you are using CICS, these can include options you want to specify to the CICS command translator.

If you specify options in this field, separate them by commas. You can continue listing options on the next line, but the total length of the option list can be no more than 70 bytes.

Fields 6 through 15 let you select the function to perform and to choose whether to show the DB2I panels for the functions you select. Use Y for YES, or N for NO.

If you are willing to accept default values for all the steps, enter N under Display panel? for all the other preparation panels listed.

To make changes to the default values, entering Y under Display panel? for any panel you want to see. DB2I then displays each of the panels that you request. After all the panels display, DB2 proceeds with the steps involved in preparing your program to run.

Variables for all functions used during program preparation are maintained separately from variables entered from the DB2I Primary Option Menu. For example, the bind plan variables you enter on the Program Preparation panel are saved separately from those on any Bind Plan panel that you reach from the Primary Option Menu.

6 CHANGE DEFAULTS

Lets you specify whether to change the DB2I defaults. Enter Y in the Display panel? field next to this option; otherwise enter N. Minimally, you should specify your subsystem identifier and programming language on the Defaults panel.

7 PL/I MACRO PHASE

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel to control the PL/I macro phase by entering PL/I options in the OPTIONS field of that panel. That panel also displays for options COMPILE OR ASSEMBLE, LINK, and RUN.

This field applies to PL/I programs only. If your program is not a PL/I program or does not use the PL/I macro processor, specify N in the Perform function field for this option, which sets the Display panel? field to the default N.

8 PRECOMPILE

Lets you specify whether to display the Precompile panel. To see this panel enter Y in the Display panel? field next to this option; otherwise enter N.

9 CICS COMMAND TRANSLATION

Lets you specify whether to use the CICS command translator. This field applies to CICS programs only.

IMS and TSO: If you run under TSO or IMS, ignore this step; this allows the Perform function field to default to N.

CICS: If you are using CICS and have precompiled your program, you must translate your program using the CICS command translator.

The command translator does not have a separate DB2I panel. You can specify translation options on the Other Options field of the DB2 Program Preparation panel, or in your source program if it is not an assembler program.

Because you specified a CICS run time environment, the Perform function column defaults to Y. Command translation takes place automatically after you precompile the program.

10 BIND PACKAGE

Lets you specify whether to display the Bind Package panel. To see it, enter Y in the Display panel? field next to this option; otherwise, enter N.

11 BIND PLAN

Lets you specify whether to display the Bind Plan panel. To see it, enter Y in the Display panel? field next to this option; otherwise, enter N.

12 COMPILE OR ASSEMBLE

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel. To see this panel enter Y in the Display panel? field next to this option; otherwise, enter N.

13 PRELINK

Lets you use the prelink utility to make your C, C++, or Enterprise COBOL for z/OS program reentrant. This utility concatenates compile-time initialization information from one or more text decks into a single initialization unit. To use the utility, enter Y in the Display panel? field next to this option; otherwise, enter N. If you request this step, then you must also request the compiler step and the link-edit step.

14 LINK

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel. To see it, enter Y in the Display panel? field next to this option; otherwise, enter N. If you specify Y in the Display panel? field for the COMPILE OR ASSEMBLE option, you do not need to make any changes to this field; the panel displayed for COMPILE OR ASSEMBLE is the same as the panel displayed for LINK. You can make the changes you want to affect the link-edit step at the same time you make the changes to the compiler step.

15 RUN

Lets you specify whether to run your program. The RUN option is available only if you specify TSO or CAF for RUN TIME ENVIRONMENT.

If you specify Y in the Display panel? field for the COMPILE OR ASSEMBLE or LINK option, you can specify N in this field, because the panel displayed for COMPILE OR ASSEMBLE and for LINK is the same as the panel displayed for RUN.

IMS and CICS: IMS and CICS programs cannot run using DB2I. If you are using IMS or CICS, use N in these fields.

TSO and batch: If you are using TSO and want to run your program, you must enter Y in the Perform function column next to this option. You can

also indicate that you want to specify options and values to affect the running of your program, by entering Y in the Display panel column.

Pressing ENTER takes you to the first panel in the series you specified, in this example to the DB2I Defaults panel. If, at any point in your progress from panel to panel, you press the END key, you return to this first panel, from which you can change your processing specifications. Asterisks (*) in the Display panel? column of rows 7 through 14 indicate which panels you have already examined. You can see a panel again by writing a Y over an asterisk.

Related reference:

“Bind Package panel” on page 993

“Bind Plan panel” on page 996

“DB2I Defaults Panel 1”

“Defaults for Bind Package and Defaults for Rebind Package panels” on page 998

“Defaults for Bind Plan and Defaults for Rebind Plan panels” on page 1001

“Precompile panel” on page 990

“Program Preparation: Compile, Link, and Run panel” on page 1005

 [DSNH \(TSO CLIST\) \(DB2 Commands\)](#)

 [Language Environment Programming Guide \(z/OS Language Environment Programming Guide\)](#)

DB2I Defaults Panel 1

DB2I Defaults Panel 1 lets you change many of the system default values that were set at DB2 installation time.

The following figure shows the fields that affect the processing of the other DB2I panels.

```
DSNEOP01                DB2I DEFAULTS PANEL 1
COMMAND ==> _

Change defaults as desired:

 1 DB2 NAME ..... ==> DSN          (Subsystem identifier)
 2 DB2 CONNECTION RETRIES ==> 0      (How many retries for DB2 connection)
 3 APPLICATION LANGUAGE ==> IBMC    (ASM, C, CPP, IBMCOB, FORTRAN, PLI)
 4 LINES/PAGE OF LISTING ==> 60      (A number from 5 to 999)
 5 MESSAGE LEVEL ..... ==> I        (Information, Warning, Error, Severe)
 6 SQL STRING DELIMITER ==> DEFAULT  (DEFAULT, ' or ")
 7 DECIMAL POINT ..... ==> .        (. or ,)
 8 STOP IF RETURN CODE >= ==> 8      (Lowest terminating return code)
 9 NUMBER OF ROWS ..... ==> 20      (For ISPF Tables)
10 AS USER ..... ==>                (User ID to associate with trusted connection)
```

Figure 55. DB2I Defaults Panel 1

The following explains the fields on DB2I Defaults Panel 1.

1 DB2 NAME

Lets you specify the DB2 subsystem that processes your DB2I requests. If you specify a different DB2 subsystem, its identifier displays in the SSID (subsystem identifier) field located at the top, right side of your screen. The default is DSN.

2 DB2 CONNECTION RETRIES

Lets you specify the number of additional times to attempt to connect to DB2, if DB2 is not up when the program issues the DSN command. The program preparation process does not use this option.

Use a number from 0 to 120. The default is 0. Connections are attempted at 30-second intervals.

3 APPLICATION LANGUAGE

Lets you specify the default programming language for your application program. You can specify any of the following languages:

ASM

For High Level Assembler/z/OS

C For C language

CPP

For C++

IBMCOB

For Enterprise COBOL for z/OS. This option is the default.

FORTRAN

For VS Fortran

PLI

For PL/I

If you specify IBMCOB, DB2 prompts you for more COBOL defaults on panel DSNEOP02. See "DB2I Defaults Panel 2" on page 989.

You cannot specify FORTRAN for IMS or CICS programs.

4 LINES/PAGE OF LISTING

Lets you specify the number of lines to print on each page of listing or SPUFI output. The default is 60.

5 MESSAGE LEVEL

Lets you specify the lowest level of message to return to you during the BIND phase of the preparation process. Use:

I For all information, warning, error, and severe error messages

W For warning, error, and severe error messages

E For error and severe error messages

S For severe error messages only

6 SQL STRING DELIMITER

Lets you specify the symbol used to delimit a string in SQL statements in COBOL programs. This option is valid only when the application language is IBMCOB. Use:

DEFAULT

To use the default defined at installation time

' For an apostrophe

" For a quotation mark

7 DECIMAL POINT

Lets you specify how your host language source program represents decimal separators and how SPUFI displays decimal separators in its output. Use a comma (,) or a period (.). The default is a period (.).

8 STOP IF RETURN CODE >=

Lets you specify the smallest value of the return code (from precompile, compile, link-edit, or bind) that will prevent later steps from running. Use:

4 To stop on warnings and more severe errors.

8 To stop on errors and more severe errors. The default is 8.

9 NUMBER OF ROWS

Lets you specify the default number of input entry rows to generate on the initial display of ISPF panels. The number of rows with non-blank entries determines the number of rows that appear on later displays.

10 AS USER

Lets you specify a user ID to associate with the trusted connection for the current DB2I session.

DB2 establishes the trusted connection for the user that you specify if the following conditions are true:

- The primary authorization ID that DB2 obtains after running the connection exit is allowed to use the trusted connection without authentication.
- The security label, if defined either implicitly or explicitly in the trusted context for the user, is defined in RACF for the user.

After DB2 establishes the trusted connection, the primary authorization ID, any secondary authorization IDs, any role, and any security label that is associated with the user ID that is specified in the AS USER field are used for the trusted connection. DB2 uses this security label to verify multilevel security for the user.

If the primary authorization ID that is associated with the user ID that is specified in the AS USER field is not allowed to use the trusted connection or requires authentication information, the connection request fails. If DB2 cannot verify the security label, the connection request also fails.

The value that you enter in this field is retained only for the length of the DB2I session. The field is reset to blank when you exit DB2I.

Suppose that the default programming language is PL/I and the default number of lines per page of program listing is 60. Your program is in COBOL, so you want to change field 3, APPLICATION LANGUAGE. You also want to print 80 lines to the page, so you need to change field 4, LINES/PAGE OF LISTING, as well. Figure 55 on page 987 shows the entries that you make in DB2I Defaults Panel 1 to make these changes. In this case, pressing ENTER takes you to DB2 Defaults Panel 2.

DB2I Defaults Panel 2

After you press Enter on the DB2I Defaults Panel 1, the DB2I Defaults Panel 2 is displayed. If you chose IBMCOB as the language on the DB2I Defaults Panel 1, three fields are displayed. Otherwise, only the first field is displayed.

The following figure shows the DB2I Defaults Panel 2 when IBMCOB is selected.


```

DSNEOP02                DB2I DEFAULTS PANEL 2
COMMAND ==>_

Change defaults as desired:

1 DB2I JOB STATEMENT:  (Optional if your site has a SUBMIT exit)
  ==> //USRT001A JOB (ACCOUNT),'NAME'
  ==> /**
  ==> /**
  ==> /**

COBOL DEFAULTS:                (For IBMCOB)
2 COBOL STRING DELIMITER ==> DEFAULT (DEFAULT, ' or ")
3 DBCS SYMBOL FOR DCLGEN ==> G      (G/N - Character in PIC clause)

```

Figure 56. DB2I Defaults Panel 2

1 DB2I JOB STATEMENT

Lets you change your default job statement. Specify a job control statement, and optionally, a JOBLIB statement to use either in the background or the EDITJCL program preparation environment. Use a JOBLIB statement to specify run time libraries that your application requires. If your program has a SUBMIT exit routine, DB2 uses that routine. If that routine builds a job control statement, you can leave this field blank.

2 COBOL STRING DELIMITER

Lets you specify the symbol used to delimit a string in a COBOL statement in a COBOL application. Use:

DEFAULT

To use the default defined at installation time

- ' For an apostrophe
- " For a quotation mark

Leave this field blank to accept the default value.

3 DBCS SYMBOL FOR DCLGEN

Lets you enter either G (the default) or N, to specify whether DCLGEN generates a picture clause that has the form PIC G(*n*) DISPLAY-1 or PIC N(*n*).

Leave this field blank to accept the default value.

Pressing ENTER takes you to the next panel you specified on the DB2 Program Preparation panel, in this case, to the Precompile panel.

Precompile panel

After you set the DB2I defaults, you can precompile your application. You can reach the Precompile panel by specifying it as a part of the program preparation process from the DB2 Program Preparation panel. Or you can reach it directly from the DB2I Primary Option Menu.

The way you choose to reach the panel determines the default values of the fields it contains. The following figure shows the Precompile panel.


```

DSNETP01                                PRECOMPILE                                SSID: DSN
COMMAND ==>_

Enter precompiler data sets:
1 INPUT DATA SET .... ==> SAMPLEPG.COBOL
2 INCLUDE LIBRARY ... ==> SRCLIB.DATA

3 DSNAME QUALIFIER .. ==> TEMP              (For building data set names)
4 DBRM DATA SET ..... ==>

Enter processing options as desired:
5 WHERE TO PRECOMPILE ==> FOREGROUND (FOREGROUND, BACKGROUND, or EDITJCL)
6 VERSION ..... ==>
7 OTHER OPTIONS ..... ==>
(BLANK, VERSION, or AUTO)

```

Figure 57. The Precompile panel. Specify the include library, if any, that your program should use, and any other options you need.

The following explains the functions on the Precompile panel, and how to enter the fields for preparing to precompile.

1 INPUT DATA SET

Lets you specify the data set name of the source program and SQL statements to precompile.

If you reached this panel through the DB2 Program Preparation panel, this field contains the data set name specified there. You can override it on this panel.

If you reached this panel directly from the DB2I Primary Option Menu, you must enter the data set name of the program you want to precompile. The data set name can include a member name. If you do not enclose the data set name with apostrophes, a standard TSO prefix (user ID) qualifies the data set name.

2 INCLUDE LIBRARY

Lets you enter the name of a library containing members that the precompiler should include. These members can contain output from DCLGEN. If you do not enclose the name in apostrophes, a standard TSO prefix (user ID) qualifies the name.

You can request additional INCLUDE libraries by entering DSNH CLIST parameters of the form PnLIB(dsname), where n is 2, 3, or 4) on the OTHER OPTIONS field of this panel or on the OTHER DSNH OPTIONS field of the Program Preparation panel.

3 DSNAME QUALIFIER

Lets you specify a character string that qualifies temporary data set names during precompile. Use any character string from 1 to 8 characters in length that conforms to normal TSO naming conventions.

If you reached this panel through the DB2 Program Preparation panel, this field contains the data set name qualifier specified there. You can override it on this panel.

If you reached this panel from the DB2I Primary Option Menu, you can either specify a DSNAME QUALIFIER or let the field take its default value, TEMP.

IMS and TSO: For IMS and TSO programs, DB2 stores the precompiled source statements (to pass to the compiler or assemble step) in a data set named *tsoprefix.qualifier.suffix*. A data set named *tsoprefix.qualifier.PCLIST* contains the precompiler print listing.

For programs prepared in the background or that use the PREPARATION ENVIRONMENT option EDITJCL (on the DB2 Program Preparation panel), a data set named *tsoprefix.qualifier.CNTL* contains the program preparation JCL.

In these examples, *tsoprefix* represents the prefix TSO assigns, often the same as the authorization ID. *qualifier* represents the value entered in the DSNAME QUALIFIER field. *suffix* represents the output name, which is one of the following: COBOL, FORTRAN, C, PLI, ASM, DECK, CICSIN, OBJ, or DATA. In the Precompile Panel that is shown above, the data set *tsoprefix.TEMP.COBOL* contains the precompiled source statements, and *tsoprefix.TEMP.PCLIST* contains the precompiler print listing. If data sets with these names already exist, then DB2 deletes them.

CICS: For CICS programs, the data set *tsoprefix.qualifier.suffix* receives the precompiled source statements in preparation for CICS command translation.

If you do not plan to do CICS command translation, the source statements in *tsoprefix.qualifier.suffix*, are ready to compile. The data set *tsoprefix.qualifier.PCLIST* contains the precompiler print listing.

When the precompiler completes its work, control passes to the CICS command translator. Because there is no panel for the translator, translation takes place automatically. The data set *tsoprefix.qualifier.CXLIST* contains the output from the command translator.

4 DBRM DATA SET

Lets you name the DBRM library data set for the precompiler output. The data set can also include a member name.

When you reach this panel, the field is blank. When you press ENTER, however, the value contained in the DSNAME QUALIFIER field of the panel, concatenated with *DBRM*, specifies the DBRM data set: *qualifier.DBRM*.

You can enter another data set name in this field only if you allocate and catalog the data set before doing so. This is true even if the data set name that you enter corresponds to what is otherwise the default value of this field.

The precompiler sends modified source code to the data set *qualifier.host*, where *host* is the language specified in the APPLICATION LANGUAGE field of DB2I Defaults panel 1.

5 WHERE TO PRECOMPILE

Lets you indicate whether to precompile in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job.

If you reached this panel from the DB2 Program Preparation panel, the field contains the preparation environment specified there. You can override that value if you want.

If you reached this panel directly from the DB2I Primary Option Menu, you can either specify a processing environment or allow this field to take its default value. Use:

BACKGROUND to immediately precompile the program with the values you specify in these panels.

BACKGROUND to create and immediately submit to run a file containing a DSNH CLIST using the JOB control statement from either DB2I Defaults Panel 2 or your site's SUBMIT exit. The file is saved. EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

6 VERSION

Lets you specify the version of the program and its DBRM. If the version contains the maximum number of characters permitted (64), you must enter each character with no intervening blanks from one line to the next. This field is optional.

7 OTHER OPTIONS

Lets you enter any option that the DSNH CLIST accepts, which gives you greater control over your program. The DSNH options you specify in this field override options specified on other panels. The option list can continue to the next line, but the total length of the list can be no more than 70 bytes.

Related reference:

 [DSNH \(TSO CLIST\) \(DB2 Commands\)](#)

Bind Package panel

The Bind Package panel is the first of two DB2I panels that request information about how you want to bind a package.

You can reach the Bind Package panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. If you enter the Bind Package panel from the Program Preparation panel, many of the Bind Package entries contain values from the Primary and Precompile panels. Figure 58 shows the Bind Package panel.

```

DSNEBP07          BIND PACKAGE          SSID: DSN

COMMAND ==>_

Specify output location and collection names:
 1 LOCATION NAME ..... ==>          (Defaults to local)
 2 COLLECTION-ID ..... ==>          (Required)
Specify package source (DBRM or COPY):
 3 DBRM:          COPY:  ==> DBRM    (Specify DBRM or COPY)
 4 MEMBER or COLLECTION-ID ==>
 5 PASSWORD or PACKAGE-ID .. ==>
 6 LIBRARY or VERSION ..... ==>
                                     (Blank, or COPY version-id)
 7 ..... -- OPTIONS ..... ==>      (COMPOSITE or COMMAND)
Enter options as desired:
 8 CHANGE CURRENT DEFAULTS? ==> NO   (NO or YES)
 9 ENABLE/DISABLE CONNECTIONS? ==> NO (NO or YES)
10 OWNER OF PACKAGE (AUTHID).. ==>   (Leave blank for primary ID)
11 QUALIFIER ..... ==>              (Leave blank for OWNER)
12 ACTION ON PACKAGE ..... ==> REPLACE (ADD or REPLACE)
13 INCLUDE PATH? ..... ==> NO       (NO or YES)
14 REPLACE VERSION ..... ==>        (Replacement version-id)

```

Figure 58. The Bind Package panel

The following information explains the functions on the Bind Package panel and how to fill the necessary fields in order to bind your program.

1 LOCATION NAME

Lets you specify the system at which to bind the package. You can use from 1 to 16 characters to specify the location name. The location name must be defined in the catalog table SYSIBM.LOCATIONS. The default is the local DBMS.

2 COLLECTION-ID

Lets you specify the collection the package is in. You can use from 1 to 18 characters to specify the collection, and the first character must be alphabetic.

3 DBRM: COPY:

Lets you specify whether you are creating a new package (DBRM) or making a copy of a package that already exists (COPY). Use:

DBRM

To create a new package. You must specify values in the LIBRARY, PASSWORD, and MEMBER fields.

COPY

To copy an existing package. You must specify values in the COLLECTION-ID and PACKAGE-ID fields. (The VERSION field is optional.)

4 MEMBER or COLLECTION-ID

MEMBER (for new packages): If you are creating a new package, this option lets you specify the DBRM to bind. You can specify a member name from 1 to 8 characters. The default name depends on the input data set name.

- If the input data set is partitioned, the default name is the member name of the input data set specified in the INPUT DATA SET NAME field of the DB2 Program Preparation panel.
- If the input data set is sequential, the default name is the second qualifier of this input data set.

COLLECTION-ID (for copying a package): If you are copying a package, this option specifies the collection ID that contains the original package. You can specify a collection ID from 1 to 18 characters, which must be different from the collection ID specified on the PACKAGE ID field.

5 PASSWORD or PACKAGE-ID

PASSWORD (for new packages): If you are creating a new package, this lets you enter password for the library you list in the LIBRARY field. You can use this field only if you reached the Bind Package panel directly from the DB2 Primary Option Menu.

PACKAGE-ID (for copying packages): If you are copying a package, this option lets you specify the name of the original package. You can enter a package ID from 1 to 8 characters.

6 LIBRARY or VERSION

LIBRARY (for new packages): If you are creating a new package, this lets you specify the names of the libraries that contain the DBRMs specified on the MEMBER field for the bind process. Libraries are searched in the order specified and must in the catalog tables.

VERSION (for copying packages): If you are copying a package, this option lets you specify the version of the original package. You can specify a version ID from 1 to 64 characters.

7 OPTIONS

Lets you specify which bind options DB2 uses when you issue BIND PACKAGE with the COPY option. Specify:

- **COMPOSITE (default)** to cause DB2 to use any options you specify in the BIND PACKAGE command. For all other options, DB2 uses the options of the copied package.
- **COMMAND** to cause DB2 to use the options you specify in the BIND PACKAGE command. For all other options, DB2 uses the following values:
 - For a local copy of a package, DB2 uses the defaults for the local DB2 subsystem.
 - For a remote copy of a package, DB2 uses the defaults for the server on which the package is bound.

8 CHANGE CURRENT DEFAULTS?

Lets you specify whether to change the current defaults for binding packages. If you enter YES in this field, you see the Defaults for Bind Package panel as your next step. You can enter your new preferences there; for instructions, see “Defaults for Bind Package and Defaults for Rebind Package panels” on page 998.

9 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 64 on page 1003) that lets you specify whether various system connections are valid for this application. You can specify connection names to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

10 OWNER OF PACKAGE (AUTHID)

Lets you specify the primary authorization ID of the owner of the new package. That ID is the name owning the package, and the name associated with all accounting and trace records produced by the package.

The owner must have the privileges required to run SQL statements contained in the package.

The default is the primary authorization ID of the bind process.

11 QUALIFIER

Lets you specify the default schema for unqualified tables, views, indexes, and aliases. You can specify a schema name from 1 to 8 characters. The default is the authorization ID of the package owner.

12 ACTION ON PACKAGE

Lets you specify whether to replace an existing package or create a new one. Use:

REPLACE (default) to replace the package named in the PACKAGE-ID field if it already exists, and add it if it does not. (Use this option if you are changing the package because the SQL statements in the program changed. If only the SQL environment changes but not the SQL statements, you can use REBIND PACKAGE.)

ADD to add the package named in the PACKAGE-ID field, only if it does not already exist.

13 INCLUDE PATH?

Indicates whether you will supply a list of schema names that DB2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. The default is NO. If you specify YES, DB2 displays a panel in which you specify the names of schemas for DB2 to search.

14 REPLACE VERSION

Lets you specify whether to replace a specific version of an existing package or create a new one. If the package and the version named in the PACKAGE-ID and VERSION fields already exist, you must specify REPLACE. You can specify a version ID from 1 to 64 characters. The default version ID is that specified in the VERSION field.

Bind Plan panel

The Bind Plan panel is the first of two DB2I panels that request information about how you want to bind an application plan.

Like the Precompile panel, you can reach the Bind Plan panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. You must have an application plan, even if you bind your application to packages; this panel also follows the Bind Package panels.

If you enter the Bind Plan panel from the Program Preparation panel, many of the Bind Plan entries contain values from the Primary and Precompile panels.

```
DSNEBP02          BIND PLAN          SSID: DSN
COMMAND ==> _

Enter primary package list:
 1 LOCATION NAME ..... ==>          (Defaults to local)
 2 COLLECTION ID ..... ==>          (Required)
 3 PACKAGE ID ..... ==>             (Package ID or *)
 4 ADDITIONAL PACKAGE LISTS .. ==> NO (YES to include more packages)

Enter options as desired:
 5 PLAN NAME ..... ==>              (Required to create a plan)
 6 CHANGE CURRENT DEFAULTS?.. ==> NO (NO or YES)
 7 ENABLE/DISABLE CONNECTIONS? ==> NO (NO or YES)
 8 OWNER OF PLAN (AUTHID) .... ==>   (Leave blank for your primary ID)
 9 QUALIFIER ..... ==>              (For tables, views, and aliases)
10 CACHESIZE ..... ==> 0             (Blank, or value 0-4096)
11 ACTION ON PLAN ..... ==> REPLACE (REPLACE or ADD)
12 RETAIN EXECUTION AUTHORITY. ==> NO (YES to retain user list)
13 CURRENT SERVER ..... ==>         (Location name)
14 INCLUDE PATH? ..... ==> NO       (NO or YES)
```

Figure 59. The Bind Plan panel

The following explains the functions on the Bind Plan panel and how to fill the necessary fields in order to bind your program.

1 LOCATION NAME

Lets you specify the remote system where the package that is named in the PACKAGE ID field is bound. The location name must be defined in the catalog table SYSIBM.LOCATIONS. The default is the local DBMS.

2 COLLECTION ID

Lets you specify the collection that includes the package that is to be bound into the plan.

3 PACKAGE ID

Lets you specify the name of the package that is to be bound into the plan.

4 ADDITIONAL PACKAGE LISTS

Lets you include a list of additional packages in the plan. If you specify YES, a separate panel displays, where you must enter the package location, collection name, and package name for each package to include in the plan. This list is optional.

5 PLAN NAME

Lets you name the application plan to create. You can specify a name from 1 to 8 characters, and the first character must be alphabetic. If there are no errors, the bind process prepares the plan and enters its description into the EXPLAIN table.

If you reached this panel through the DB2 Program Preparation panel, the default for this field depends on the value you entered in the INPUT DATA SET NAME field of that panel.

If you reached this panel directly from the DB2 Primary Option Menu, you must include a plan name if you want to create an application plan. The default name for this field depends on the input data set:

- If the input data set is partitioned, the default name is the member name.
- If the input data set is sequential, the default name is the second qualifier of the data set name.

6 CHANGE CURRENT DEFAULTS?

Lets you specify whether to change the current defaults for binding plans. If you enter YES in this field, you see the Defaults for Bind Plan panel as your next step. You can enter your new preferences there.

7 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 64 on page 1003) that lets you specify whether various system connections are valid for this application. You can specify connection names to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

8 OWNER OF PLAN (AUTHID)

Lets you specify the primary authorization ID of the owner of the new plan. That ID is the name owning the plan, and the name associated with all accounting and trace records produced by the plan.

The owner must have the privileges required to run SQL statements contained in the plan.

9 QUALIFIER

Lets you specify the default schema for unqualified tables, views, and aliases. You can specify a schema name from 1 to 8 characters, which must conform to the rules for SQL identifiers. If you leave this field blank, the default qualifier is the authorization ID of the plan owner.

10 CACHESIZE

Lets you specify the size (in bytes) of the authorization cache. Valid values

are in the range 0 to 4096. Values that are not multiples of 256 round up to the next highest multiple of 256. A value of 0 indicates that DB2 does not use an authorization cache. The default is 1024.

Each concurrent user of a plan requires 8 bytes of storage, with an additional 32 bytes for overhead.

11 ACTION ON PLAN

Lets you specify whether this is a new or changed application plan. Use:

REPLACE (default) to replace the plan named in the PLAN NAME field if it already exists, and add the plan if it does not exist.

ADD to add the plan named in the PLAN NAME field, only if it does not already exist.

12 RETAIN EXECUTION AUTHORITY

Lets you choose whether or not those users with the authority to bind or run the existing plan are to keep that authority over the changed plan. This applies only when you are replacing an existing plan.

If the plan ownership changes and you specify YES, the new owner grants BIND and EXECUTE authority to the previous plan owner.

If the plan ownership changes and you do not specify YES, then everyone but the new plan owner loses EXECUTE authority (but not BIND authority), and the new plan owner grants BIND authority to the previous plan owner.

13 CURRENT SERVER

Lets you specify the initial server to receive and process SQL statements in this plan. You can specify a name from 1 to 16 characters, which you must previously define in the catalog table SYSIBM.LOCATIONS.


If you specify a remote server, DB2 connects to that server when the first SQL statement executes. The default is the name of the local DB2 subsystem.

14 INCLUDE PATH?

Indicates whether you will supply a list of schema names that DB2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. The default is NO. If you specify YES, DB2 displays a panel in which you specify the names of schemas for DB2 to search.

When you finish making changes to this panel, press ENTER to go to the second of the program preparation panels, Program Prep: Compile, Link, and Run.

Related tasks:

 [Caching authorization IDs for plans \(Managing Security\)](#)

Related reference:

“Defaults for Bind Plan and Defaults for Rebind Plan panels” on page 1001

 [BIND and REBIND options for packages and plans \(DB2 Commands\)](#)

Defaults for Bind Package and Defaults for Rebind Package panels

These DB2I panels lets you change your defaults for BIND PACKAGE and REBIND PACKAGE options.

On the following panel, enter new defaults for binding a package.

```
DSNEBP10          DEFAULTS FOR BIND PACKAGE          SSID: DSN
COMMAND ==> _
----- Use the UP/DOWN keys to access all options -----
                                                    More:  +

Change default options as necessary:

 1 ISOLATION LEVEL ..... ==>      (CS, RR, RS, UR, or NC)
 2 VALIDATION TIME ..... ==>      (RUN or BIND)
 3 RESOURCE RELEASE TIME .. ==>    (COMMIT or DEALLOCATE)
 4 EXPLAIN PATH SELECTION .. ==>   (NO or YES)
 5 DATA CURRENCY ..... ==>       (NO or YES)
 6 PARALLEL DEGREE ..... ==>      (1 or ANY)
 7 SQLERROR PROCESSING .... ==>    (NOPACKAGE or CONTINUE)
 8 REOPTIMIZE FOR INPUT VARS ==>   (ALWAYS, NONE, ONCE, or AUTO)
 9 DEFER PREPARE ..... ==>        (NO OR YES)
10 KEEP DYN SQL PAST COMMIT ==>    (NO or YES)
11 APPLICATION ENCODING ... ==>    (Blank, ASCII, EBCDIC,
                                     UNICODE, or ccsid)
12 OPTIMIZATION HINT ..... ==>    (Blank or 'hint-id')
13 IMMEDIATE WRITE ..... ==>      (YES, NO)
14 DYNAMIC RULES ..... ==>        (RUN, BIND, DEFINE, or INVOKE)
15 DBPROTOCOL ..... ==>           (blank or DRDA)
16 ACCESS PATH REUSE ..... ==>    NONE (ERROR or NONE)
17 ACCESS PATH COMPARISON .. ==>   NONE (ERROR, NONE, or WARN)
-----
PRESS: ENTER to continue  UP/DOWN to scroll  RETURN to EXIT
```

Figure 60. The Defaults for Bind Package panel

On the following panel, enter new defaults for rebinding a package.

With a few minor exceptions, the options on this panel are the same as the options for the defaults for rebinding a package. However, the defaults for REBIND PACKAGE are different from those shown in the preceding figure, and you can specify SAME in any field to specify the values used the last time the package was bound. For rebinding, the default value for all fields is SAME.

```

DSNEBP11          DEFAULTS FOR REBIND PACKAGE          SSID: DSN
COMMAND ==> _
----- Use the UP/DOWN keys to access all options -----
More:      +

Change default options as necessary:

 1 ISOLATION LEVEL ..... ==>      (SAME, CS, RR, RS, UR, or NC)
 2 PLAN VALIDATION TIME ... ==>    (SAME, RUN, or BIND)
 3 RESOURCE RELEASE TIME ... ==>   (SAME, DEALLOCATE, COMMIT,
    OR INHERITFROMPLAN)
 4 EXPLAIN PATH SELECTION .. ==>   (SAME, NO, or YES)
 5 DATA CURRENCY ..... ==>       (SAME, NO, or YES)
 6 PARALLEL DEGREE ..... ==>      (SAME, 1 or ANY)
 7 REOPTIMIZE FOR INPUT VARS ==>   (SAME, ALWAYS, NONE, ONCE, AUTO)
 8 DEFER PREPARE ..... ==>        (SAME, NO, YES,
    OR INHERITFROMPLAN)
 9 KEEP DYN SQL PAST COMMIT ==>    (SAME, NO, or YES)
10 APPLICATION ENCODING ... ==>    (SAME, Blank, ASCII, EBCDIC,
    UNICODE, or ccsid)
11 OPTIMIZATION HINT ..... ==>    (Blank or 'hint-id')
12 IMMEDIATE WRITE ..... ==>     (SAME, NO, YES,
    OR INHERITFROMPLAN)
13 DBPROTOCOL ..... ==>          (blank or DRDA)
14 DYNAMIC RULES ..... ==>       (SAME, RUN, BIND,
    DEFINERUN, DEFINEBIND,
    INVOKERUN or INVOKEBIND)
15 PLAN MANAGEMENT ..... ==>     (DEFAULT, BASIC, EXTENDED, OFF)
16 ACCESS PATH REUSE ..... ==>    (DEFAULT, ERROR, NONE, or WARN)
17 ACCESS PATH COMPARISON .. ==>  (DEFAULT, ERROR, NONE, or WARN)
18 ACCESS PATH RETAIN DUPS . ==>  (DEFAULT, NO, OR YES)
-----
PRESS:  ENTER to continue  UP/DOWN to scroll  RETURN to EXIT

```

Figure 61. The Defaults for Rebind Package panel

The following table lists the fields on the Defaults for Bind Package and Defaults for Rebind Package panels, and the corresponding bind and rebind options.

Table 160. Defaults for Bind Package and Defaults for Rebind Package panel fields and corresponding bind or rebind options


Field name	Bind or rebind option
ACCESS PATH COMPARISON	APCOMPARE
ACCESS PATH RETAIN DUPS	APRETAINDUP
ACCESS PATH REUSE	APREUSE
APPLICATION ENCODING	ENCODING
DATA CURRENCY	CURRENTDATA
DBPROTOCOL	DBPROTOCOL
DEFER PREPARE	DEFER and NODEFER
DYNAMIC RULES	DYNAMICRULES
EXPLAIN PATH SELECTION	EXPLAIN
IMMEDIATE WRITE	IMMEDWRITE
ISOLATION LEVEL	ISOLATION
KEEP DYN SQL PAST COMMIT	KEEPDYNAMIC
OPTIMIZATION HINT	OPTHINT
PARALLEL DEGREE	DEGREE
PLAN MANAGEMENT	PLANMGMT
REOPTIMIZE FOR INPUT VARS	REOPT
RESOURCE RELEASE TIME	RELEASE

Table 160. Defaults for Bind Package and Defaults for Rebind Package panel fields and corresponding bind or rebind options (continued)

Field name	Bind or rebind option
SQLERROR PROCESSING	SQLERROR
VALIDATION TIME and PLAN VALIDATION TIME	VALIDATE

Related concepts:

“DYNAMICRULES bind option” on page 959

 Parallel processing (DB2 Performance)

 Investigating SQL performance by using EXPLAIN (DB2 Performance)

Related tasks:

“Setting the isolation level of SQL statements in a REXX program” on page 445

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

Defaults for Bind Plan and Defaults for Rebind Plan panels

These DB2I panels let you change your defaults for BIND PLAN and REBIND PLAN options.

On the following panel, enter new defaults for binding a plan.

```

DSNEBP10                DEFAULTS FOR BIND PLAN                SSID: DSN
COMMAND ==>>>

Change default options as necessary:

 1 ISOLATION LEVEL ..... ==>> RR          (RR, RS, CS, or UR)
 2 VALIDATION TIME ..... ==>> RUN         (RUN or BIND)
 3 RESOURCE RELEASE TIME ... ==>> COMMIT   (COMMIT or DEALLOCATE)
 4 EXPLAIN PATH SELECTION .. ==>> NO      (NO or YES)
 5 DATA CURRENCY ..... ==>> NO          (NO or YES)
 6 PARALLEL DEGREE ..... ==>> 1          (1 or ANY)
 7 RESOURCE ACQUISITION TIME ==>> USE     (USE or ALLOCATE)
 8 REOPTIMIZE FOR INPUT VARS ==>> NONE    (ALWAYS, NONE, ONCE, AUTO)
 9 DEFER PREPARE ..... ==>> NO          (NO or YES)
10 KEEP DYN SQL PAST COMMIT. ==>> NO      (NO or YES)
11 APPLICATION ENCODING ... ==>>         (Blank,ASCII,EBCDIC,UNICODE,
                                     or ccsid)
12 OPTIMIZATION HINT ..... ==>>         (Blank or 'hint-id')
13 IMMEDIATE WRITE ..... ==>> NO        (YES, NO)
14 DYNAMIC RULES ..... ==>> RUN         (RUN or BIND)
15 SQLRULES..... ==>> DB2              (DB2 or STD)
16 DISCONNECT ..... ==>> EXPLICIT      (EXPLICIT, AUTOMATIC,
                                     or CONDITIONAL)

```

Figure 62. The Defaults for Bind Plan panel

On the following panel, enter new defaults for rebinding a plan.

```

DSNEBP11          DEFAULTS FOR REBIND PLAN          SSID: DSN
COMMAND ==>>>

Change default options as necessary:

 1 ISOLATION LEVEL ..... ==>>          (SAME, RR, RS, CS, or UR)
 2 PLAN VALIDATION TIME .... ==>>      (SAME, RUN, or BIND)
 3 RESOURCE RELEASE TIME ... ==>>      (SAME, DEALLOCATE, or COMMIT)
 4 EXPLAIN PATH SELECTION .. ==>>      (SAME, NO, or YES)
 5 DATA CURRENCY ..... ==>>          (SAME, NO, or YES)
 6 PARALLEL DEGREE ..... ==>>          (SAME, 1 or ANY)
 7 REOPTIMIZE FOR INPUT VARS ==>>      (SAME, ALWAYS, NONE, ONCE, AUTO)
 8 DEFER PREPARE ..... ==>>           (SAME, NO, or YES)
 9 KEEP DYN SQL PAST COMMIT. ==>>      (SAME, NO, or YES)
10 APPLICATION ENCODING ... ==>>      (SAME,Blank,ASCII,EBCDIC,
                                     UNICODE, or ccsid)

11 OPTIMIZATION HINT ..... ==>>      (SAME, 'hint-id')
12 IMMEDIATE WRITE ..... ==>>        (SAME, YES, NO)
13 SQLRULES ..... ==>>              (SAME, DB2 or STD)
14 DYNAMIC RULES ..... ==>>         (SAME, RUN, or BIND)
15 RESOURCE ACQUISITION TIME ==>>    (SAME, ALLOCATE, or USE)
16 DISCONNECT ..... ==>>           (SAME, EXPLICIT, AUTOMATIC,
                                     or CONDITIONAL)

```

Figure 63. The Defaults for Rebind Plan panel

The following table lists the fields on the Defaults for Bind Package and Defaults for Rebind Package, and the corresponding bind and rebind options.

Table 161. Defaults for Bind Plan and Defaults for Rebind Plan panel fields and corresponding bind or rebind options

Field name	Bind or rebind option
APPLICATION ENCODING	ENCODING
DATA CURRENCY	CURRENTDATA
DBPROTOCOL	DBPROTOCOL
DEFER PREPARE	DEFER and NODEFER
DISCONNECT	DISCONNECT
DYNAMIC RULES	DYNAMICRULES
EXPLAIN PATH SELECTION	EXPLAIN
IMMEDIATE WRITE	IMMEDWRITE
ISOLATION LEVEL	ISOLATION
KEEP DYN SQL PAST COMMIT	KEEPDYNAMIC
OPTIMIZATION HINT	OPTHINT
PARALLEL DEGREE	DEGREE
REOPTIMIZE FOR INPUT VARS	REOPT
RESOURCE ACQUISITION TIME	ACQUIRE
RESOURCE RELEASE TIME	RELEASE
VALIDATION TIME and PLAN VALIDATION TIME	VALIDATE


Related concepts:

“DYNAMICRULES bind option” on page 959

 Parallel processing (DB2 Performance)

 Investigating SQL performance by using EXPLAIN (DB2 Performance)

Related tasks:

 Caching authorization IDs for plans (Managing Security)

“Setting the isolation level of SQL statements in a REXX program” on page 445

“Specifying the rules that apply to SQL behavior at run time” on page 972

System Connection Types panel

The System Connection Types panel lets you specify which types of connections can use a plan or package.

This panel displays if you enter YES for ENABLE/DISABLE CONNECTIONS? on the Bind or Rebind Package or Plan panels. For the Bind or Rebind Package panel, the REMOTE option does not display as it does in the following panel.

```
DSNEBP13  SYSTEM CONNECTION TYPES FOR BIND ...          SSID: DSN
COMMAND ==>

Select system connection types to be Enabled/Disabled:

 1  ENABLE ALL CONNECTION TYPES? ==>   (* to enable all types)
or
 2  ENABLE/DISABLE SPECIFIC CONNECTION TYPES ==>   (E/D)

BATCH ..... ==>   (Y/N)          SPECIFY CONNECTION NAMES?
DB2CALL ..... ==>   (Y/N)
RRSAF ..... ==>   (Y/N)
CICS ..... ==>   (Y/N)          ==> N (Y/N)
IMS ..... ==>   (Y/N)
DLIBATCH ..... ==>   (Y/N)          ==> N (Y/N)
IMSBMP ..... ==>   (Y/N)          ==> N (Y/N)
IMSMPP ..... ==>   (Y/N)          ==> N (Y/N)
REMOTE ..... ==>   (Y/N)          ==> N (Y/N)
```

Figure 64. The System Connection Types panel

To enable or disable connection types (that is, allow or prevent the connection from running the package or plan), enter the following information.

1 ENABLE ALL CONNECTION TYPES?

Lets you enter an asterisk (*) to enable all connections. After that entry, you can ignore the rest of the panel.

2 ENABLE/DISABLE SPECIFIC CONNECTION TYPES

Lets you specify a list of types to enable or disable; you cannot enable some types and disable others in the same operation. If you list types to enable, enter E; that disables all other connection types. If you list types to disable, enter D; that enables all other connection types.

For each connection type that follows, enter Y (yes) if it is on your list, N (no) if it is not. The connection types are:

- **BATCH** for a TSO connection
- **DB2CALL** for a CAF connection
- **RRSAF** for an RRSAF connection
- **CICS** for a CICS connection
- **IMS** for all IMS connections: DLIBATCH, IMSBMP, and IMSMPP
- **DLIBATCH** for a DL/I Batch Support Facility connection
- **IMSBMP** for an IMS connection to a BMP region
- **IMSMPP** for an IMS connection to an MPP or IFP region
- **REMOTE** for remote location names and LU names

On the system command line, you can use:

END Saves all entered variables, exits the table, and continues to process.

CANCEL

Discards all entered variables, terminates processing, and returns to the previous panel.

SAVE Saves all entered variables and remains in the table.

In the special command area, you can use:

Inn Insert *nn* lines after this one.

Dnn Delete this and the following lines for *nn* lines.

Rnn Repeat this line *nn* number of times.

The default for *nn* is 1.

When you finish with a list panel, specify END to same the current panel values and continue processing.

Program Preparation: Compile, Link, and Run panel

The Compile, Link, and Run panel lets you perform the last two steps in the program preparation process (compile and link-edit). This panel also lets you perform the PL/I MACRO PHASE for programs that require this option.

For TSO programs, the panel also lets you run programs.

```
DSNEPP02    PROGRAM PREP: COMPILE, PRELINK, LINK, AND RUN    SSID: DSN
COMMAND ==>_

Enter compiler or assembler options:
1 INCLUDE LIBRARY ==> SRCLIB.DATA
2 INCLUDE LIBRARY ==>
3 OPTIONS ..... ==> NUM, OPTIMIZE, ADV

Enter linkage editor options:
4 INCLUDE LIBRARY ==> SAMPLIB.COBOL
5 INCLUDE LIBRARY ==>
6 INCLUDE LIBRARY ==>
7 LOAD LIBRARY .. ==> RUNLIB.LOAD
8 PRELINK OPTIONS ==>
9 LINK OPTIONS... ==>

Enter run options:
10 PARAMETERS ... ==> D01, D02, D03/
11 SYSIN DATA SET ==> TERM
12 SYSPRINT DS ... ==> TERM
```

Figure 66. The Program Preparation: Compile, Link, and Run panel

1,2 INCLUDE LIBRARY

Lets you specify up to two libraries containing members for the compiler to include. The members can also be output from DCLGEN. You can leave these fields blank. There is no default.

3 OPTIONS

Lets you specify compiler, assembler, or PL/I macro processor options. You can also enter a list of compiler or assembler options by separating entries with commas, blanks, or both. You can leave these fields blank. There is no default.

4,5,6 INCLUDE LIBRARY

Lets you enter the names of up to three libraries containing members for the linkage editor to include. You can leave these fields blank. There is no default.

7 LOAD LIBRARY

Lets you specify the name of the library to hold the load module. The default value is RUNLIB.LOAD.

If the load library specified is a PDS, and the input data set is a PDS, the member name specified in INPUT DATA SET NAME field of the Program Preparation panel is the load module name. If the input data set is sequential, the second qualifier of the input data set is the load module name.

You must complete this field if you request LINK or RUN on the Program Preparation panel.

8 PRELINK OPTIONS

Lets you enter a list of prelinker options. Separate items in the list with commas, blanks, or both. You can leave this field blank. There is no default.

The prelink utility applies only to programs using C, C++, and Enterprise COBOL for z/OS.

9 LINK OPTIONS

Lets you enter a list of link-edit options. Separate items in the list with commas, blanks, or both.

To prepare a program that uses 31-bit addressing and runs above the 16-megabyte line, specify the following link-edit options: AMODE=31, RMODE=ANY.

10 PARAMETERS

Lets you specify a list of parameters you want to pass either to your host language run time processor, or to your application. Separate items in the list with commas, blanks, or both. You can leave this field blank.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Use a slash (/) to separate the options for your run time processor from those for your program.

- For PL/I and Fortran, run time processor parameters must appear on the left of the slash, and the application parameters must appear on the right.
run time processor parameters / application parameters
- For COBOL, reverse this order. run time processor parameters must appear on the right of the slash, and the application parameters must appear on the left.
- For assembler and C, there is no supported run time environment, and you need not use a slash to pass parameters to the application program.

11 SYSIN DATA SET

Lets you specify the name of a SYSIN (or in Fortran, FT05F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

12 SYSPRINT DS

Lets you specify the names of a SYSPRINT (or in Fortran, FT06F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Your application could need other data sets besides SYSIN and SYSPRINT. If so, remember to catalog and allocate them before you run your program.

When you press ENTER after entering values in this panel, DB2 compiles and link-edits the application. If you specified in the DB2 Program Preparation panel that you want to run the application, DB2 also runs the application.

Related reference:

 Language Environment Programming Guide (z/OS Language Environment Programming Guide)

DB2I panels that are used to rebind and free plans and packages

A set of DB2I panels lets you bind, rebind, or free packages.

Table 162 describes additional panels that you can use to Rebind and Free packages and plans. It also describes the Run panel, which you can use to run application programs that have already been prepared.

Table 162. DB2I panels used to rebind and free plans and packages and used to Run application programs

Panel	Panel description
"Bind/Rebind/Free Selection panel" on page 1008	The BIND/REBIND/FREE panel lets you select the BIND, REBIND, or FREE, PLAN, PACKAGE, or TRIGGER PACKAGE process that you need.
"Rebind Package panel" on page 1009	The Rebind Package panel lets you change options when you rebind a package.
"Rebind Trigger Package panel" on page 1011	The Rebind Trigger Package panel lets you change options when you rebind a trigger package.
"Rebind Plan panel" on page 1013	The Rebind Plan panel lets you change options when you rebind an application plan.
"Free Package panel" on page 1014	The Free Package panel lets you change options when you free a package.
"Free Plan panel" on page 1015	The Free Plan panel lets you change options when you free an application plan.
"DB2I Run panel" on page 1018	The Run panel lets you start an application program. You should use this panel if you have already prepared the program and you only want to run it. You can also run a program by using the "Program Prep: Compile, Prelink, Link, and Run" panel.

Bind/Rebind/Free Selection panel

The Bind/Rebind/Free selection panel lets choose whether to bind, rebind, or free plans and packages.

```
DSNEBP01          BIND/REBIND/FREE          SSID: DSN
COMMAND ==>>>_

Select one of the following and press ENTER:

 1 BIND PLAN          (Add or replace an application plan)
 2 REBIND PLAN        (Rebind existing application plan or plans)
 3 FREE PLAN          (Erase application plan or plans)
 4 BIND PACKAGE        (Add or replace a package)
 5 REBIND PACKAGE      (Rebind existing package or packages)
 6 REBIND TRIGGER PACKAGE (Rebind existing package or packages)
 7 FREE PACKAGE        (Erase a package or packages)
```

Figure 67. The Bind/Rebind/Free selection panel

This panel lets you select the process you need.

1 BIND PLAN

Lets you build an application plan. You must have an application plan to allocate DB2 resources and support SQL requests during run time. If you select this option, the Bind Plan panel displays. For more information, see “Bind Plan panel” on page 996.

2 REBIND PLAN

Lets you rebuild an application plan when changes to it affect the plan but the SQL statements in the program are the same. For example, you should rebind when you change authorizations, create a new index that the plan uses, or use RUNSTATS. If you select this option, the Rebind Plan panel displays. For more information, see “Rebind Plan panel” on page 1013.

3 FREE PLAN

Lets you delete plans from DB2. If you select this option, the Free Plan panel displays. For more information, see “Free Plan panel” on page 1015.

4 BIND PACKAGE

Lets you build a package. If you select this option, the Bind Package panel displays. For more information, see “Bind Package panel” on page 993.

5 REBIND PACKAGE

Lets you rebuild a package when changes to it affect the package but the SQL statements in the program are the same. For example, you should rebind when you change authorizations, create a new index that the package uses, or use RUNSTATS. If you select this option, the Rebind Package panel displays. For more information, see “Rebind Package panel” on page 1009.

6 REBIND TRIGGER PACKAGE

Lets you rebuild a trigger package when you need to change options for the package. When you execute CREATE TRIGGER, DB2 binds a trigger package using a set of default options. You can use REBIND TRIGGER PACKAGE to change those options. For example, you can use REBIND TRIGGER PACKAGE to change the isolation level for the trigger package.

If you select this option, the Rebind Trigger Package panel displays. For more information, see “Rebind Trigger Package panel” on page 1011.

7 FREE PACKAGE

Lets you delete a specific version of a package, all versions of a package, or whole collections of packages from DB2. If you select this option, the Free Package panel displays. For more information, see “Free Package panel” on page 1014.

Rebind Package panel

The Rebind Package panel is the first of two panels that you use to rebind a package. This panel lets you specify options for rebinding the package.

The following figure shows the rebind package options.

```

DSNEBP08                REBIND PACKAGE                SSID: DSN
COMMAND ==>_

 1 Rebind all local packages ==>          (* to rebind all packages)

or
  Enter package name(s) to be rebound:
 2 LOCATION NAME ..... ==>              (Defaults to local)
 3 COLLECTION-ID ..... ==>              (Required)
 4 PACKAGE-ID ..... ==>                 (Required)
 5 VERSION-ID ..... ==>
 6 ADDITIONAL PACKAGES? ..... ==>      (*, Blank, (), or version-id)
                                       (Yes to include more packages)

Enter options as desired ..... ==>
 7 CHANGE CURRENT DEFAULTS?... ==>      (NO or YES)
 8 OWNER OF PACKAGE (AUTHID).. ==>      (SAME, new OWNER)
 9 QUALIFIER ..... ==>                  (SAME, new QUALIFIER)
10 ENABLE/DISABLE CONNECTIONS? ==>     (NO or YES)
11 INCLUDE PATH? ..... ==>             (SAME, DEFAULT, or YES)

```

Figure 68. The Rebind Package panel

This panel lets you choose options for rebinding a package.

1 Rebind all local packages

Lets you rebind all packages on the local DBMS. To do so, place an asterisk (*) in this field; otherwise, leave it blank.

2 LOCATION NAME

Lets you specify where to bind the package. If you specify a location name, you should use from 1 to 16 characters, and you must have defined it in the catalog table SYSIBM.LOCATIONS.

3 COLLECTION-ID

Lets you specify the collection of the package to rebind. You must specify a collection ID from 1 to 8 characters, or an asterisk (*) to rebind all collections in the local DB2 system. You cannot use the asterisk to rebind a remote collection.

4 PACKAGE-ID

Lets you specify the name of the package to rebind. You must specify a package ID from 1 to 8 characters, or an asterisk (*) to rebind all packages in the specified collections in the local DB2 system. You cannot use the asterisk to rebind a remote package.

5 VERSION-ID

Lets you specify the version of the package to rebind. You must specify a version ID from 1 to 64 characters, or an asterisk (*) to rebind all versions in the specified collections and packages in the local DB2 system. You cannot use the asterisk to rebind a remote version.

6 ADDITIONAL PACKAGES?

Lets you indicate whether to name more packages to rebind. Use YES to specify more packages on an additional panel, described on “Panels for entering lists of values” on page 1004. The default is NO.

7 CHANGE CURRENT DEFAULTS?

Lets you indicate whether to change the binding defaults. Use:

NO (default) to retain the binding defaults of the previous package.

YES to change the binding defaults from the previous package. For information about the defaults for binding packages, see “Defaults for Bind Package and Defaults for Rebind Package panels” on page 998.

8 OWNER OF PACKAGE (AUTHID)

Lets you change the authorization ID for the package owner. The owner must have the required privileges to execute the SQL statements in the package. The default is the existing package owner.

9 QUALIFIER

Lets you specify the default schema for all unqualified table names, views, indexes, and aliases in the package. You can specify a schema name from 1 to 8 characters, which must conform to the rules for the SQL short identifier. The default is the existing qualifier name.

10 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 64 on page 1003) that lets you specify whether various system connections are valid for this application.

The default is the values used for the previous package.


11 INCLUDE PATH?

Indicates which one of the following actions you want to perform:

- Request that DB2 uses the same schema names as when the package was bound for resolving unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose SAME to perform this action. This is the default.
- Supply a list of schema names that DB2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose YES to perform this action.
- Request that DB2 resets the SQL path to SYSIBM, SYSFUN, SYSPROC, and the package owner. Choose DEFAULT to perform this action.

If you specify YES, DB2 displays a panel in which you specify the names of schemas for DB2 to search.

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

Rebind Trigger Package panel

The Rebind Trigger Package panel specifies options for rebinding a trigger package.

The following figure shows those options.

```
DSNEBP19                REBIND TRIGGER PACKAGE                SSID: DSN
COMMAND ==>_

1 Rebind all trigger packages ==>                (* to rebind all packages)

or
Enter trigger package name(s) to be rebound:
2 LOCATION NAME ..... ==>                (Defaults to local)
3 COLLECTION-ID (SCHEMA NAME) ==>                (Required)
4 PACKAGE-ID (TRIGGER NAME).. ==>                (Required)

Enter options as desired ..... ==>
5 ISOLATION LEVEL ..... ==> SAME                (SAME, RR, RS, CS, UR, or NC)
6 RESOURCE RELEASE TIME ..... ==> SAME                (SAME, DEALLOCATE, or COMMIT)
7 EXPLAIN PATH SELECTION .... ==> SAME                (SAME, NO, or YES)
8 DATA CURRENCY ..... ==> SAME                (SAME, NO, or YES)
9 IMMEDIATE WRITE OPTION .... ==> SAME                (SAME, NO, YES)
10 PLAN MANAGEMENT ..... ==> DEFAULT                (DEFAULT, BASIC, EXTENDED, OFF)
11 ACCESS PATH REUSE ..... ==> DEFAULT                (DEFAULT, ERROR, or NONE)
12 ACCESS PATH COMPARISON .... ==> DEFAULT                (DEFAULT, ERROR, NONE, or WARN)
13 ACCESS PATH RETAIN DUPS ... ==> DEFAULT                (DEFAULT, NO, or YES)
```

Figure 69. The Rebind Trigger Package panel

This panel lets you choose options for rebinding a trigger package.

1 Rebind all trigger packages

Lets you rebind all packages on the local DBMS. To do so, place an asterisk (*) in this field; otherwise, leave it blank.

2 LOCATION NAME

Lets you specify where to bind the trigger package. If you specify a location name, you should use from 1 to 16 characters, and you must have defined it in the catalog table SYSIBM.LOCATIONS.

3 COLLECTION-ID (SCHEMA NAME)

Lets you specify the collection of the trigger package to rebind. You must specify a collection ID from 1 to 8 characters, or an asterisk (*) to rebind all collections in the local DB2 system. You cannot use the asterisk to rebind a remote collection.

4 PACKAGE-ID

Lets you specify the name of the trigger package to rebind. You must specify a package ID from 1 to 8 characters, or an asterisk (*) to rebind all trigger packages in the specified collections in the local DB2 system. You cannot use the asterisk to rebind a remote trigger package.

5 ISOLATION LEVEL

Lets you specify how far to isolate your application from the effects of other running applications. The default is the value used for the old trigger package.

6 RESOURCE RELEASE TIME

Lets you specify COMMIT or DEALLOCATE to tell when to release locks on resources. The default is that used for the old trigger package.

7 EXPLAIN PATH SELECTION

Lets you specify YES or NO for whether to obtain EXPLAIN information about how SQL statements in the package execute. The default is the value used for the old trigger package.

The bind process inserts information into the table *owner.PLAN_TABLE*, where *owner* is the authorization ID of the plan or package owner. If you defined *owner.DSN_STATEMNT_TABLE*, DB2 also inserts information about the cost of statement execution into that table. If you specify YES in this field and BIND in the VALIDATION TIME field, and if you do not correctly define PLAN_TABLE, the bind fails.

8 DATA CURRENCY

Lets you specify YES or NO for whether you need data currency for ambiguous cursors opened at remote locations. The default is the value used for the old trigger package.

Data is current if the data within the host structure is identical to the data within the base table. Data is always current for local processing.

9 IMMEDIATE WRITE OPTION

Specifies when DB2 writes the changes for updated group buffer pool-dependent pages. This field applies only to a data sharing environment. The values that you can specify are:

SAME Choose the value of IMMEDIATE WRITE that you specified when you bound the trigger package. SAME is the default.

NO Write the changes at or before phase 1 of the commit process. If the transaction is rolled back later, write the additional changes that are caused by the rollback at the end of the abort process.

PH1 is equivalent to NO.

YES Write the changes immediately after group buffer pool-dependent pages are updated.

10 PLAN MANAGEMENT

Specifies the PLANMGMT option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

11 ACCESS PATH REUSE

Specifies the APREUSE option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.


12 ACCESS PATH COMPARISON

Specifies the APCOMPARE option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

13 ACCESS PATH RETAIN DUPS

Specifies the APRETAINDUP option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

Related reference:

 BIND and REBIND options for packages and plans (DB2 Commands)

Rebind Plan panel

The Rebind Plan panel is the first of two panels that you use to rebind a plan. This panel lets you specify options for rebinding the plan.

The following figure shows the rebind plan options.

```
DSNEBP03                REBIND PLAN                SSID: DSN
COMMAND ==>_

Enter plan name(s) to be rebound:
 1 PLAN NAME ..... ==>          (* to rebind all plans)
 2 ADDITIONAL PLANS? ..... ==> NO (Yes to include more plans)

Enter options as desired:
 3 CHANGE CURRENT DEFAULTS?... ==> NO      (NO or YES)
 4 OWNER OF PLAN (AUTHID)..... ==> SAME   (SAME, new OWNER)
 5 QUALIFIER ..... ==> SAME              (SAME, new QUALIFIER)
 6 CACHESIZE ..... ==> SAME              (SAME, or value 0-4096)
 7 ENABLE/DISABLE CONNECTIONS? ==> NO     (NO or YES)
 8 INCLUDE PACKAGE LIST?..... ==> SAME   (SAME, NO, or YES)
 9 CURRENT SERVER ..... ==>              (Location name)
10 INCLUDE PATH? ..... ==> SAME         (SAME, DEFAULT, or YES)
```

Figure 70. The Rebind Plan panel

This panel lets you specify options for rebinding your plan.

1 PLAN NAME

Lets you name the application plan to rebind. You can specify a name from 1 to 8 characters, and the first character must be alphabetic. Do not begin the name with DSN, because it could create name conflicts with DB2. If there are no errors, the bind process prepares the plan and enters its description into the EXPLAIN table.

If you leave this field blank, the bind process occurs but produces no plan.

2 ADDITIONAL PLANS?

Lets you indicate whether to name more plans to rebind. Use YES to specify more plans on an additional panel, described at “Panels for entering lists of values” on page 1004. The default is NO.

3 CHANGE CURRENT DEFAULTS?

Lets you indicate whether to change the binding defaults. Use:
NO (default) to retain the binding defaults of the previous plan.
YES to change the binding defaults from the previous plan.

4 OWNER OF PLAN (AUTHID)

Lets you change the authorization ID for the plan owner. The owner must have the required privileges to execute the SQL statements in the plan. The default is the existing plan owner.

5 QUALIFIER

Lets you specify the default schema for all unqualified table names, views, indexes, and aliases in the plan. You can specify a schema name from 1 to 8 characters, which must conform to the rules for the SQL identifier. The default is the authorization ID.

6 CACHESIZE

Lets you specify the size (in bytes) of the authorization cache. Valid values are in the range 0 to 4096. Values that are not multiples of 256 round up to the next highest multiple of 256. A value of 0 indicates that DB2 does not use an authorization cache. The default is the cache size specified for the previous plan.

Each concurrent user of a plan requires 8 bytes of storage, with an additional 32 bytes for overhead.

7 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this plan. This is valid only for rebinding on your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 64 on page 1003) that lets you specify whether various system connections are valid for this application.

The default is the values used for the previous plan.

8 INCLUDE PACKAGE LIST?

Lets you include a list of collections and packages in the plan. If you specify YES, a separate panel displays on which you must enter the package location, collection name, and package name for each package to include in the plan (see “Panels for entering lists of values” on page 1004). This field can either add a package list to a plan that did not have one, or replace an existing package list.

You can specify a location name from 1 to 16 characters, a collection ID from 1 to 18 characters, and a package ID from 1 to 8 characters. Separate two or more package list parameters with a comma. If you specify a location name, it must be in the catalog table SYSIBM.LOCATIONS. The default location is the package list used for the previous plan.

9 CURRENT SERVER

Lets you specify the initial server to receive and process SQL statements in this plan. You can specify a name from 1 to 16 characters, which you must previously define in the catalog table SYSIBM.LOCATIONS.

If you specify a remote server, DB2 connects to that server when the first SQL statement executes. The default is the name of the local DB2 subsystem.

10 INCLUDE PATH?


Indicates which one of the following actions you want to perform:

- Request that DB2 uses the same schema names as when the plan was bound for resolving unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose SAME to perform this action. This is the default.
- Supply a list of schema names that DB2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose YES to perform this action.
- Request that DB2 resets the SQL path to SYSIBM, SYSFUN, SYSPROC, and the plan owner. Choose DEFAULT to perform this action.

If you specify YES, DB2 displays a panel in which you specify the names of schemas for DB2 to search.

Related reference:

“Defaults for Bind Plan and Defaults for Rebind Plan panels” on page 1001

 BIND and REBIND options for packages and plans (DB2 Commands)

Free Package panel

The DB2I Free Package panel is the first of two panels through which you can specify options for freeing an application package.

The following figure shows the free package options.

```
DSNEBP18          FREE PACKAGE          SSID: DSN
COMMAND ==>_

 1 Free ALL packages ..... ==>      (* to free authorized packages)

or

  Enter package name(s) to be freed:
 2 LOCATION NAME ..... ==>          (Defaults to local)
 3 COLLECTION-ID ..... ==>          (Required)
 4 PACKAGE-ID ..... ==>            (* to free all packages)
 5 VERSION-ID ..... ==>            (*, Blank, (), or version-id)
 6 ADDITIONAL PACKAGES?..... ==>   (Yes to include more packages)
 7 PLAN MANAGEMENT SCOPE ..... ==> ALL (ALL or INACTIVE)
```

Figure 71. The Free Package panel

This panel lets you specify options for erasing packages.

1 Free ALL packages

Lets you free (erase) all packages for which you have authorization or to which you have BINDAGENT authority. To do so, place an asterisk (*) in this field; otherwise, leave it blank.

2 LOCATION NAME

Lets you specify the location name of the DBMS to free the package. You can specify a name from 1 to 16 characters.

3 COLLECTION-ID

Lets you specify the collection from which you want to delete packages for which you own or have BINDAGENT privileges. You can specify a name from 1 to 18 characters, or an asterisk (*) to free all collections in the local DB2 system. You cannot use the asterisk to free a remote collection.

4 PACKAGE-ID

Lets you specify the name of the package to free. You can specify a name from 1 to 8 characters, or an asterisk (*) to free all packages in the specified collections in the local DB2 system. You cannot use the asterisk to free a remote package. The name you specify must be in the DB2 catalog tables.

5 VERSION-ID

Lets you specify the version of the package to free. You can specify an identifier from 1 to 64 characters, or an asterisk (*) to free all versions of the specified collections and packages in the local DB2 system. You cannot use the asterisk to free a remote version.

6 ADDITIONAL PACKAGES?

Lets you indicate whether to name more packages to free. Use YES to specify more packages on an additional panel, described in "Panels for entering lists of values" on page 1004. The default is NO.

7 PLAN MANAGEMENT SCOPE

Specifies whether DB2 frees all copies of the package, or only the inactive previous and original copies. This value corresponds to the PLANMGMTSCOPE option. The default value is ALL.

Free Plan panel

The DB2I Free Plan panel is the first of two panels through which you can specify options for freeing an application plan.

Figure 72 shows the free plan options.

```
DSNEBP09          FREE PLAN          SSID: DSN
COMMAND ==>_

Enter plan name(s) to be freed:
1 PLAN NAME ..... ==>          (* to free all authorized plans)
2 ADDITIONAL PLANS? .... ==>    (Yes to include more plans)
```

Figure 72. The Free Plan panel

This panel lets you specify options for freeing plans.

1 PLAN NAME

Lets you name the application plan to delete from DB2. Use an asterisk to free all plans for which you have BIND authority. You can specify a name from 1 to 8 characters, and the first character must be alphabetic.

If there are errors, the free process terminates for that plan and continues with the next plan.

2 ADDITIONAL PLANS?

Lets you indicate whether to name more plans to free. Use YES to specify more plans on an additional panel, described in “Panels for entering lists of values” on page 1004. The default is NO.

Chapter 18. Running an application on DB2 for z/OS

You can run your application after you have processed the SQL statements, compiled and link-edited the application, and bound the application.

At run time, DB2 verifies that the information in the application plan and its associated packages is consistent with the corresponding information in the DB2 catalog. If any destructive changes, such as DROP or REVOKE, occur (either to the data structures that your application accesses or to the binder's authority to access those data structures), DB2 automatically rebinds packages or the plan as needed.

Establishing a test environment: This topic describes how to design a test data structure and how to fill tables with test data.

CICS Before you run an application, ensure that the following two conditions are met:

- The corresponding entries in the SNT and RACF control areas authorize your application to run.
- The program and its transaction code are defined in the CICS CSD.

The system administrator is responsible for these functions.

DSN command processor

The DSN command processor is a TSO command processor that runs in TSO foreground or under TSO in a JES-initiated batch environment.

It uses the TSO attachment facility to access DB2. The DSN command processor provides an alternative method for running programs that access DB2 in a TSO environment.

When you run an application by using the DSN command processor, that application can run in a trusted connection if DB2 finds a matching trusted context.

You can use the DSN command processor implicitly during program development for functions such as:

- Using the declarations generator (DCLGEN)
- Running the BIND, REBIND, and FREE subcommands on DB2 plans and packages for your program
- Using SPUFI (SQL Processor Using File Input) to test some of the SQL functions in the program

The DSN command processor runs with the TSO terminal monitor program (TMP). Because the TMP runs in either foreground or background, DSN applications run interactively or as batch jobs.

The DSN command processor can provide these services to a program that runs under it:

- Automatic connection to DB2
- Attention key support
- Translation of return codes into error messages

Limitations of the DSN command processor:

When using DSN services, your application runs under the control of DSN. Because TSO executes the ATTACH macro to start DSN, and DSN executes the ATTACH macro to start a part of itself, your application gains control that is two task levels below TSO.

Because your program depends on DSN to manage your connection to DB2:

- If DB2 is down, your application cannot begin to run.
- If DB2 terminates, your application also terminates.
- An application can use only one plan.

If these limitations are too severe, consider having your application use the call attachment facility or Resource Recovery Services attachment facility. For more information about these attachment facilities, see “Call attachment facility” on page 77 and “Resource Recovery Services attachment facility” on page 108.

DSN return code processing: At the end of a DSN session, register 15 contains the highest value that is placed there by any DSN subcommand that is used in the session or by any program that is run by the RUN subcommand. Your run time environment might format that value as a return code. The value **does not**, however, originate in DSN.

DB2I Run panel

The DB2I Run panel lets you start an application program that can contain SQL statements.

You can reach the Run panel only through the DB2I Primary Options Menu. You can accomplish the same task using the “Program Preparation: Compile, Link, and Run” panel. You should use this panel if you have already prepared the program and simply want to run it. Figure 73 shows the run options.

```
DSNERP01          RUN          SSID: DSN
COMMAND ==>_

Enter the name of the program you want to run:
 1 DATA SET NAME ==>
 2 PASSWORD..... ==>      (Required if data set is password protected)

Enter the following as desired:
 3 PARAMETERS .. ==>
 4 PLAN NAME ... ==>      (Required if different from program name)
 5 WHERE TO RUN ==>      (FOREGROUND, BACKGROUND, or EDITJCL)
```

Figure 73. The Run panel

This panel lets you run existing application programs.

1 DATA SET NAME

Lets you specify the name of the partitioned data set that contains the load module. If the module is in a data set that the operating system can find, you can specify the member name only. There is no default.

If you do not enclose the name in apostrophes, a standard TSO prefix (user ID) and suffix (.LOAD) is added.

2 PASSWORD

Lets you specify the data set password if needed. The RUN processor does

not check whether you need a password. If you do not enter a required password, your program does not run.

3 PARAMETERS

Lets you specify a list of parameters you want to pass either to your host language run time processor, or to your application. You should separate items in the list with commas, blanks, or both. You can leave this field blank.

Use a slash (/) to separate the options for your run time processor from those for your program.

- For PL/I and Fortran, run time processor parameters must appear on the left of the slash, and the application parameters must appear on the right.
run time processor parameters / application parameters
- For COBOL, reverse this order. run time processor parameters must appear on the right of the slash, and the application parameters must appear on the left.
- For assembler and C, there is no supported run time environment, and you need not use the slash to pass parameters to the application program.

4 PLAN NAME

Lets you specify the name of the plan to which the program is bound. The default is the member name of the program.

5 WHERE TO RUN

Lets you indicate whether to run in the foreground or background. You can also specify EDITJCL, in which case you are able to edit the job control statement before you run the program. Use:

FOREGROUND to immediately run the program in the foreground with the specified values.

BACKGROUND to create and immediately submit to run a file containing a DSNH CLIST using the JOB control statement from either DB2I Defaults Panel 2 or your site's SUBMIT exit. The program runs in the background.

EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it. The program runs in the background.

Running command processors

To run a command processor (CP), use the following commands from the TSO ready prompt or as a TSO TMP:

```
DSN SYSTEM (DB2-subsystem-name)  
RUN CP PLAN (plan-name)
```

The RUN subcommand prompts you for more input. The end the DSN processor, use the END command.

Running a program in TSO foreground

Use the DB2I RUN panel to run a program in TSO foreground. Alternatively, you can issue the DSN command, followed by the RUN subcommand of DSN.

Before running the program, be sure to allocate any data sets that your program needs.

The following example shows how to start a TSO foreground application. The name of the application is SAMPPGM, and *ssid* is the system ID:

```
TSO Prompt:  READY
Enter:     DSN SYSTEM(ssid)
DSN Prompt:  DSN
Enter:     RUN PROGRAM(SAMPPGM) -
              PLAN(SAMPLAN) -
              LIB(SAMPPROJ.SAMPLIB) -
              PARS('/D01 D02 D03')
:
(Here the program runs and might prompt you for input)
DSN Prompt:  DSN
Enter:     END
TSO Prompt:  READY
```

This sequence also works in ISPF option 6. You can package this sequence in a CLIST. DB2 does not support access to multiple DB2 subsystems from a single address space.

The PARS keyword of the RUN subcommand enables you to pass parameters to the run time processor and to your application program:

```
PARMS ('/D01, D02, D03')
```

The slash (/) indicates that you are passing parameters. For some languages, you pass parameters and run time options in the form PARS(*parameters/run time-options*). An example of the PARS keyword might be:

```
PARMS ('D01, D02, D03/')
```

Check your host language publications for the correct form of the PARS option.

Running a DB2 REXX application

You run DB2 REXX applications under TSO. You do not precompile, compile, link-edit, or bind DB2 REXX applications before you run them.

In a batch environment, you might use statements like these to invoke application REXXPROG:

```
//RUNREXX EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSEXEC DD DISP=SHR,DSN=SYSADM.REXX.EXEC
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
%REXXPROG parameters
```

The SYSEXEC data set contains your REXX application, and the SYSTSIN data set contains the command that you use to invoke the application.

Invoking programs through the Interactive System Productivity Facility

You can use ISPF to invoke programs that connect to DB2 through the call attachment facility (CAF).

The ISPF/CAF sample connection manager programs (DSN8SPM and DSN8SCM) take advantage of the ISPLINK SELECT services, letting each routine make its own connection to DB2 and establish its own thread and plan.

With the same modular structure as in the previous example, using CAF is likely to provide greater efficiency by reducing the number of CLISTs. This does not mean, however, that any DB2 function executes more quickly.

Disadvantages: Compared to the modular structure using DSN, the structure using CAF is likely to require a more complex program, which in turn might require assembler language subroutines. For more information, see “Call attachment facility” on page 77.

ISPF

The Interactive System Productivity Facility (ISPF) helps you to construct and execute dialogs. DB2 includes a sample application that illustrates how to use ISPF through the call attachment facility (CAF).

Each scenario has advantages and disadvantages in terms of efficiency, ease of coding, ease of maintenance, and overall flexibility.

Using ISPF and the DSN command processor

There are some restrictions on how you make and break connections to DB2 in any structure. If you use the PGM option of ISPF SELECT, ISPF passes control to your load module by the LINK macro; if you use CMD, ISPF passes control by the ATTACH macro.

The DSN command processor permits only single task control block (TCB) connections. Take care not to change the TCB after the first SQL statement. ISPF SELECT services change the TCB if you started DSN under ISPF, so you cannot use these to pass control from load module to load module. Instead, use LINK, XCTL, or LOAD.

The following figure shows the task control blocks that result from attaching the DSN command processor below TSO or ISPF.

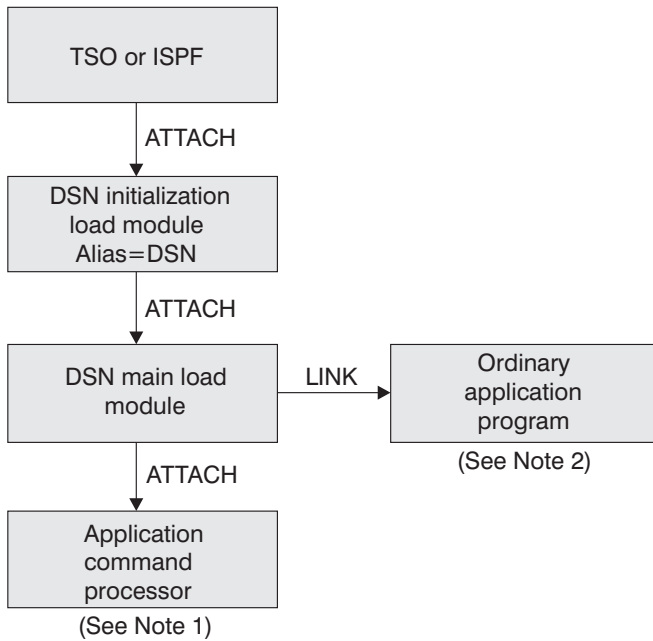


Figure 74. DSN task structure

Notes:

1. The RUN command with the CP option causes DSN to attach your program and create a new TCB.
2. The RUN command without the CP option causes DSN to link to your program.

If you are in ISPF and running under DSN, you can perform an ISPLINK to another program, which calls a CLIST. In turn, the CLIST uses DSN and another application. Each such use of DSN creates a separate unit of recovery (process or transaction) in DB2.

All such initiated DSN work units are unrelated, with regard to isolation (locking) and recovery (commit). It is possible to deadlock with yourself; that is, one unit (DSN) can request a serialized resource (a data page, for example) that another unit (DSN) holds incompatibly.

A COMMIT in one program applies only to that process. There is no facility for coordinating the processes.

Related concepts:

➡ Dynamic SQL and the ISPF/CAF application (DB2 Installation and Migration)

➡ Printing options for the sample application listings (DB2 Installation and Migration)

“DB2 sample applications” on page 1092

“DSN command processor” on page 1017

Invoking a single SQL program through ISPF and DSN

When you invoke a single SQL program through ISPF and DSN, you should first invoke ISPF, which displays the data and selection panels. When you select the program on the selection panel, ISPF calls a CLIST that runs the program.

A corresponding CLIST might contain:


```
DSN
  RUN PROGRAM(MYPROG) PLAN(MYPLAN)
END
```

The application has one large load module and one plan.

Disadvantages: For large programs of this type, you want a more modular design, making the plan more flexible and easier to maintain. If you have one large plan, you must rebind the entire plan whenever you change a module that includes SQL statements. To achieve a more modular construction when all parts of the program use SQL, consider using packages. See “DB2 program preparation overview” on page 973 You cannot pass control to another load module that makes SQL calls by using ISPLINK; rather, you must use LINK, XCTL, or LOAD and BALR.

If you want to use ISPLINK, then call ISPF to run under DSN:

```
DSN
  RUN PROGRAM(ISPF) PLAN(MYPLAN)
END
```

You then need to leave ISPF before you can start your application.

Furthermore, the entire program is dependent on DB2; if DB2 is not running, no part of the program can begin or continue to run.

Invoking multiple SQL programs through ISPF and DSN

You can break a large application into several different functions. Each function communicates through a common pool of shared variables, which is controlled by ISPF.

You might write some functions as separately compiled and loaded programs, others as EXECs or CLISTs. You can start any of those programs or functions through the ISPF SELECT service, and you can start that from a program, a CLIST, or an ISPF selection panel.

When you use the ISPF SELECT service, you can specify whether ISPF should create a new ISPF variable pool before calling the function. You can also break a large application into several independent parts, each with its own ISPF variable pool.

You can call different parts of the program in different ways. For example, you can use the PGM option of ISPF SELECT:

```
PGM(program-name) PARM(parameters)
```

Alternatively, you can use the CMD option:

```
CMD(command)
```

For a part that accesses DB2, the command can name a CLIST that starts DSN:

```
DSN
  RUN PROGRAM(PART1) PLAN(PLAN1) PARM(input from panel)
END
```

Breaking the application into separate modules makes it more flexible and easier to maintain. Furthermore, some of the application might be independent of DB2; portions of the application that do not call DB2 can run, even if DB2 is not running. A stopped DB2 database does not interfere with parts of the program that refer only to other databases.

Disadvantages: The modular application, on the whole, has to do more work. It calls several CLISTs, and each one must be located, loaded, parsed, interpreted, and executed. It also makes and breaks connections to DB2 more often than the single load module. As a result, you might lose some efficiency.

Loading and running a batch program

You can run a DL/I batch program by running module DSNMTV01, which loads your application, or by running the application program directly.

To run a program using DB2, you need a DB2 plan. The bind process creates the DB2 plan. DB2 first verifies whether the DL/I batch job step can connect to batch job DB2. Then DB2 verifies whether the application program can access DB2 and enforce user identification of batch jobs accessing DB2.

The two ways to submit DL/I batch applications to DB2 are:

- The DL/I batch procedure can run module DSNMTV01 as the application program. DSNMTV01 loads the “real” application program.
- The DL/I batch procedure can run your application program without using module DSNMTV01. To accomplish this, perform the following actions:
 - Specify SSM= in the DL/I batch procedure.
 - In the batch region of your application JCL, specify the following information:
 - MBR=*application-name*
 - SSM=*DB2 subsystem name*

Submitting a DL/I batch application using DSNMTV01: The following skeleton JCL example illustrates a COBOL application program, IVP8CP22, that runs using DB2 DL/I batch support.

- The first step uses the standard DLIBATCH IMS procedure.
- The second step shows how to use the DFSERA10 IMS program to print the contents of the DDOTV02 output data set.

```
//ISOC04 JOB 3000,IS0IR,MSGLEVEL=(1,1),NOTIFY=IS0IR,
//      MSGCLASS=T,CLASS=A
//JOBLIB DD DISP=SHR,
//      DSN=prefix.SDSNLOAD
//* *****
//*
//* THE FOLLOWING STEP SUBMITS COBOL JOB IVP8CP22, WHICH UPDATES
//* BOTH DB2 AND DL/I DATABASES.
//*
//* *****
//UPDTE EXEC DLIBATCH,DBRC=Y,LOGT=SYSDA,COND=EVEN,
// MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N//G.STEPLIB DD
//      DD
//      DD DSN=prefix.SDSNLOAD,DISP=SHR
//      DD DSN=prefix.RUNLIB.LOAD,DISP=SHR
//      DD DSN=SYS1.COB2LIB,DISP=SHR
//      DD DSN=IMS.PGMLIB,DISP=SHR
//G.DDOTV02 DD DSN=&TEMP1,DISP=(NEW,PASS,DELETE),
//      SPACE=(TRK,(1,1),RLSE),UNIT=SYSDA,
//      DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02 DD *
//      SSDQ,SYS1,DSNMIN10,,A,-,BATCH001,,IVP8CP22
/*
//*****
//*** ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET ***
//*****
//STEP3 EXEC PGM=DFSERA10,COND=EVEN
//STEPLIB DD DSN=IMS.RESLIB,DISP=SHR
```

```

//SYSPRINT DD SYSOUT=A
//SYSUT1 DD DSN=&TEMP1,DISP=(OLD,DELETE)
//SYSIN DD *
CONTROL CNTL K=000,H=8000
OPTION PRINT
/*
//

```

Submitting a DL/I batch application without using DSNMTV01: The skeleton JCL in the following example illustrates a COBOL application program, IVP8CP22, that runs using DB2 DL/I batch support.

```

//TEPCTEST JOB 'USER=ADMF001',MSGCLASS=A,MSGLEVEL=(1,1),
//          TIME=1440,CLASS=A,USER=SYSADM,PASSWORD=SYSADM
//*****
//BATCH EXEC DLIBATCH,PSB=IVP8CA,MBR=IVP8CP22,
//          BKO=Y,DBRC=N,IRLM=N,SSM=SSDQ
//*****
//SYSPRINT DD SYSOUT=A
//REPORT DD SYSOUT=*
//G.DDOTV02 DD DSN=&TEMP,DISP=(NEW,PASS,DELETE),
//          SPACE=(CYL,(10,1),RLSE),
//          UNIT=SYSDA,DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02 DD *
SSDQ,SYS1,DSNMIN10,,Q," ,DSNMTES1,,IVP8CP22
//G.SYSIN DD *
/*
//*****
//* ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET
//*****
//PRTLOG EXEC PGM=DFSER10,COND=EVEN
//STEPLIB DD DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SYSUT1 DD DSN=&TEMP,DISP=(OLD,DELETE)
//SYSIN DD *
CONTROL CNTL K=000,H=8000
OPTION PRINT
/*

```

Authorization for running a batch DL/I program

When a DL/I batch application tries to run the first SQL statement, DB2 checks whether the authorization ID has the EXECUTE privilege for the plan. DB2 uses the same ID for subsequent authorization checks and also identifies records from the accounting and performance traces.

The primary authorization ID is the value of the USER parameter on the job statement, if that is available. If that parameter is not available, the primary authorization ID is the TSO logon name if the job is submitted. Otherwise, the primary authorization ID is the IMS PSB name. In that case, however, the ID must not begin with the string "SYSADM" because this string causes the job to abnormally terminate. The batch job is rejected if you try to change the authorization ID in an exit routine.

Restarting a batch program

To restart a batch program that updates data, first run the IMS Batch Backout utility, followed by a restart job indicating the last successful checkpoint ID.

For guidelines on finding the last successful checkpoint, see "Finding the DL/I batch checkpoint ID" on page 1027.

JCL example of a batch backout: The skeleton JCL example that follows illustrates a batch backout for PSB=IVP8CA.

```
//ISOC04 JOB 3000,ISOIR,MSGLEVEL=(1,1),NOTIFY=ISOIR,
//      MSGCLASS=T,CLASS=A
//* * * * *
//*
//* BACKOUT TO LAST CHKPT.
//*          IF RC=0028 LOG WITH NO-UPDATE
//*
//* - - - - -
//BACKOUT EXEC PGM=DFSRR00,
//      PARM='DLI,DFSBB000,IVP8CA,,,,,,,,,Y,N,,Y',
//      REGION=2600K,COND=EVEN          |
//*                                     ---> DBRC ON
//STEPLIB DD DSN=IMS.RESLIB,DISP=SHR
//IMS      DD DSN=IMS.PSBLIB,DISP=SHR
//          DD DSN=IMS.DBDLIB,DISP=SHR
//*
//* IMSLOGR DD data set is required
//* IEFRDER DD data set is required
//DFSVSAMP DD *
OPTIONS,LTWA=YES
2048,7
1024,7
/*
//SYSIN DD DUMMY
/*
```

JCL example of restarting a DL/I batch job: Operational procedures can restart a DL/I batch job step for an application program using IMS XRST and symbolic CHKP calls.

You cannot restart a BMP application program in a DB2 DL/I batch environment. The symbolic checkpoint records are not accessed, causing an IMS user abend U0102.

To restart a batch job that terminated abnormally or prematurely, find the checkpoint ID for the job on the z/OS system log or from the SYSOUT listing of the failing job. Before you restart the job step, place the checkpoint ID in the CKPTID=value option of the DLIBATCH procedure, submit the job. If the default connection name is used (that is, you did not specify the connection name option in the DDITV02 input data set), the job name of the restart job must be the same as the failing job. Refer to the following skeleton example, in which the last checkpoint ID value was IVP80002:

```
//ISOC04 JOB 3000,OJALA,MSGLEVEL=(1,1),NOTIFY=OJALA,
//      MSGCLASS=T,CLASS=A
//* *****
//*
//* THE FOLLOWING STEP RESTARTS COBOL PROGRAM IVP8CP22, WHICH UPDATES
//* BOTH DB2 AND DL/I DATABASES, FROM CKPTID=IVP80002.
//*
//* *****
//RSTRT EXEC DLIBATCH,DBRC=Y,COND=EVEN,LOGT=SYSDA,
// MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N,CKPTID=IVP80002
//G.STEPLIB DD
//      DD
//      DD DSN=prefix.SDSNLOAD,DISP=SHR
//      DD DSN=prefix.RUNLIB.LOAD,DISP=SHR
//      DD DSN=SYS1.COB2LIB,DISP=SHR
//      DD DSN=IMS.PGMLIB,DISP=SHR
//* other program libraries
//* G.IEFRDER data set required
//* G.IMSLOGR data set required
```

```

//G.DDOTV02 DD DSN=&TEMP2,DISP=(NEW,PASS,DELETE),
//          SPACE=(TRK,(1,1),RLSE),UNIT=SYSDA,
//          DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02 DD *
          DB2X,SYS1,DSNMIN10,,A,-,BATCH001,,IVP8CP22
/*
//*****
//*** ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET ***
//*****
//STEP8     EXEC  PGM=DFSERA10,COND=EVEN
//STEPLIB  DD     DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT DD     SYSOUT=A
//SYSUT1   DD     DSNAME=&TEMP2,DISP=(OLD,DELETE)
//SYSIN    DD     *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
//

```

Finding the DL/I batch checkpoint ID

When an application program issues an IMS CHKP call, IMS sends the checkpoint ID to the z/OS console and the SYSOUT listing in message DFS0540I.

IMS also records the checkpoint ID in the type X'41' IMS log record. Symbolic CHKP calls also create one or more type X'18' records on the IMS log. XRST uses the type X'18' log records to reposition DL/I databases and return information to the application program.

During the commit process the application program checkpoint ID is passed to DB2. If a failure occurs during the commit process, creating an indoubt work unit, DB2 remembers the checkpoint ID. You can use the following techniques to find the last checkpoint ID:

- Look at the SYSOUT listing for the job step to find message DFS0540I, which contains the checkpoint IDs that are issued. Use the last listed checkpoint ID.
- Look at the z/OS console log to find message DFS0540I that contains the checkpoint ID that is issued for this batch program. Use the last listed checkpoint ID.
- Submit the IMS Batch Backout utility to back out the DL/I databases to the last (default) checkpoint ID. When the batch backout finishes, message DFS395I provides the last valid IMS checkpoint ID. Use this checkpoint ID on restart.
- When restarting DB2, issue the command `-DISPLAY THREAD(*) TYPE(INDOUBT)` to obtain a possible indoubt unit of work (connection name and checkpoint ID). If you restarted the application program from this checkpoint ID, the program could work because the checkpoint is recorded on the IMS log; however, the program could fail with an IMS user abend U102 because IMS did not finish logging the information before the failure. In that case, restart the application program from the previous checkpoint ID.

DB2 performs one of two actions automatically when restarted, if the failure occurs outside the indoubt period: it either backs out the work unit to the prior checkpoint, or it commits the data without any assistance. If the operator then issues the following command, no work unit information is displayed:

```
-DISPLAY THREAD(*) TYPE(INDOUBT)
```

Running stored procedures from the command line processor

As an alternative to calling a stored procedure from an application program, you can use the command line processor to invoke stored procedures.

To run a stored procedure from the command line processor:

1. Invoke the command line processor and connect to the appropriate DB2 subsystem. For more information about how to perform these tasks, see *Command line processor (DB2 Commands)*.
2. Specify the CALL statement in the form that is acceptable for the command line processor.

Related tasks:

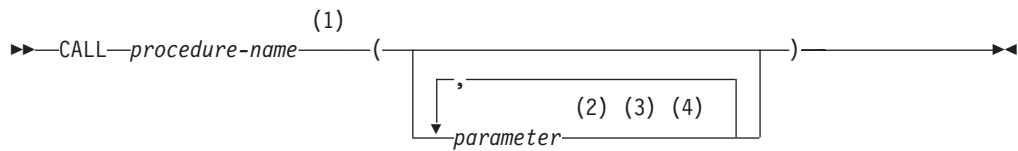
Chapter 14, "Calling a stored procedure from your application," on page 791

 [Implementing DB2 stored procedures \(DB2 Administration Guide\)](#)

Command line processor CALL statement

Use the command line processor CALL statement to invoke stored procedures from the command line processor.

Use the following syntax for the command line processor CALL statement.



Notes:

- 1 If you specify an unqualified stored procedure name, DB2 searches the schema list in the CURRENT PATH special register. DB2 searches this list for a stored procedure with the specified number of input and output parameters.
- 2 Specify a question mark (?) as a placeholder for each output parameter.
- 3 For non-numeric, BLOB, or CLOB input parameters, enclose each value in single quotation marks ('). The exception is if the data is a BLOB or CLOB value that is to be read from a file. In that case, use the notation *file://fully qualified file name*.
- 4 Specify the input and output parameters in the order that they are specified in the signature for the stored procedure.

Example: Assume that the TEST.DEPT_MEDIAN stored procedure was created with the following statement:

```
CREATE PROCEDURE TEST.DEPT_MEDIAN  
(IN DEPTNUMBER SMALLINT, OUT MEDIANSALARY INT)
```

To invoke the stored procedure from the command line processor, you can specify the following CALL statement:

```
CALL TEST.DEPT_MEDIAN(51, ?)
```

Assume that the stored procedure returns a value of 25,000. The following information is displayed by the command line processor:

Value of output parameters

Parameter Name : MEDIANSALARY
Parameter Value : 25000

Example: Suppose that stored procedure TEST.BLOBSP is defined with one input parameter of type BLOB and one output parameter. You can invoke this stored procedure from the command line processor with the following statement:

```
CALL TEST.BLOBSP(file:///tmp/photo.bmp,?)
```

The command line processor reads the contents from /tmp/photo.bmp as the input parameter. Alternatively, you can invoke this stored procedure by specifying the input parameter in the CALL statement itself, as in the following example:

```
CALL TEST.BLOBSP('abcdef',?)
```

Example of running a batch DB2 application in TSO

Most application programs that are written for the batch environment run under the TSO Terminal Monitor Program (TMP) in background mode.

The following figure shows the JCL statements that you need in order to start such a job. The list that follows explains each statement.

```
//jobname JOB USER=MY DB2ID  
//GO EXEC PGM=IKJEFT01,DYNAMNBR=20  
//STEPLIB DD DSN=prefix.SDSNEXIT,DISP=SHR  
//          DD DSN=prefix.SDSNLOAD,DISP=SHR  
:  
//SYSTSPRT DD SYSOUT=A  
//SYSTSIN DD *  
DSN SYSTEM (ssid)  
RUN PROG (SAMPPGM) -  
  PLAN (SAMPLAN) -  
  LIB (SAMPPROJ.SAMPLIB) -  
  PARS ('/D01 D02 D03')  
END  
/*
```

- The JOB option identifies this as a job card. The USER option specifies the DB2 authorization ID of the user.
- The EXEC statement calls the TSO Terminal Monitor Program (TMP).
- The STEPLIB statement specifies the library in which the DSN Command Processor load modules and DSNHDECP or a user-specified application defaults module reside. It can also reference the libraries in which user applications, exit routines, and the customized DSNHDECP module reside. The customized DSNHDECP module is created during installation.
- Subsequent DD statements define additional files that are needed by your program.
- The DSN command connects the application to a particular DB2 subsystem.
- The RUN subcommand specifies the name of the application program to run.
- The PLAN keyword specifies plan name.
- The LIB keyword specifies the library that the application should access.
- The PARS keyword passes parameters to the run time processor and the application program.
- END ends the DSN command processor.

Usage notes

- Keep DSN job steps short.
-

Recommendation: Do not use DSN to call the EXEC command processor to run CLISTs that contain ISPEXEC statements; results are unpredictable.

- If your program abends or gives you a non-zero return code, DSN terminates.
- You can use a group attachment or subgroup attachment name instead of a specific *ssid* to connect to a member of a data sharing group.

Related reference:

 Using the TSO TMP in batch mode (TSO/E User's Guide)

Example of calling applications in a command procedure

As an alternative to foreground or batch calls to an application, you can run a TSO or batch application by using a command procedure (CLIST).

The following CLIST calls a DB2 application program named MYPROG. *ssid* represents the DB2 subsystem name, or group attachment or subgroup attachment name.

```
PROC 0                                /* INVOCATION OF DSN FROM A CLIST */
  DSN SYSTEM(ssid)                   /* INVOKE DB2 SUBSYSTEM ssid */
  IF &LASTCC = 0 THEN                 /* BE SURE DSN COMMAND WAS SUCCESSFUL */
  DO                                  /* IF SO THEN DO DSN RUN SUBCOMMAND */
    DATA                             /* ELSE OMIT THE FOLLOWING: */
      RUN PROGRAM(MYPROG)
    END
  ENDDATA                             /* THE RUN AND THE END ARE FOR DSN */
  END
EXIT
```

IMS: To run a message-driven program

First, ensure that you can respond to the program's interactive requests for data and that you can recognize the expected results. Then, enter the transaction code that is associated with the program. Users of the transaction code must be authorized to run the program.

To run a non-message-driven program

CICS To run a program

First, ensure that the corresponding entries in the SNT and RACF control areas allow run authorization for your application. The system administrator is responsible for these functions.

Submit the job control statements that are needed to run the program.

Also, be sure to define to CICS the transaction code that is assigned to your program and the program itself.


Make a new copy of the program


Issue the NEWCOPY command if CICS has not been reinitialized since the program was last bound and compiled.

Chapter 19. Testing and debugging an application program on DB2 for z/OS

Depending on the situation, testing your application program might involve setting up a test environment, testing SQL statements, debugging your programs, and reading output from the precompiler.

Related tasks:

 Modeling a production environment on a test subsystem (DB2 Performance)

 Modeling your production system statistics in a test subsystem (DB2 Performance)

Designing a test data structure

When you test an application that accesses DB2 data, you should have DB2 data available for testing. To do this, you can create test tables and views.

- **Test views of existing tables:** If your application does not change a set of DB2 data and the data exists in one or more production-level tables, you might consider using a view of existing tables.
- **Test tables:** To create a test table, you need a database and table space. Talk with your DBA to make sure that a database and table spaces are available for your use.


If the data that you want to change already exists in a table, consider using the LIKE clause of CREATE TABLE. If you want others besides yourself to have ownership of a table for test purposes, you can specify a secondary ID as the owner of the table. You can do this with the SET CURRENT SQLID statement.


If your location has a separate DB2 system for testing, you can create the test tables and views on the test system and then test your program thoroughly on that system. This information assumes that you do all testing on a separate system, and that the person who created the test tables and views has an authorization ID of TEST. The table names are TEST.EMP, TEST.PROJ and TEST.DEPT.

Related concepts:

 Authorization IDs (Managing Security)

Related tasks:

 Modeling a production environment on a test subsystem (DB2 Performance)

 Modeling your production system statistics in a test subsystem (DB2 Performance)

Related reference:

 SET CURRENT SQLID (DB2 SQL)

Analyzing application data needs

To design tests of an application, you need to determine the type of data that the application uses and how the application accesses that data.

This information assumes that you do all testing on a separate system, and that the person who created the test tables and views has an authorization ID of TEST. The table names are TEST.EMP, TEST.PROJ and TEST.DEPT.

To design test tables and views, first analyze the data needs of your application.

To analyze the data needs of your application:

1. List the data that your application accesses and describe how it accesses each data item. For example, suppose that you are testing an application that accesses the DSN8A10.EMP, DSN8A10.DEPT, and DSN8A10.PROJ tables. You might record the information about the data as shown in Table 163.

Table 163. Description of the application data

Table or view name	Insert rows?	Delete rows?	Column name	Data type	Update access?
DSN8A10.EMP	No	No	EMPNO	CHAR(6)	No
			LASTNAME	VARCHAR(15)	No
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	Yes
			JOB	DECIMAL(3)	Yes
DSN8A10.DEPT	No	No	DEPTNO	CHAR(3)	No
			MGRNO	CHAR (6)	No
DSN8A10.PROJ	Yes	Yes	PROJNO	CHAR(6)	No
			DEPTNO	CHAR(3)	Yes
			RESPEMP	CHAR(6)	Yes
			PRSTAFF	DECIMAL(5,2)	Yes
			PRSTDATE	DECIMAL(6)	Yes
			PRENDATE	DECIMAL(6)	Yes

2. Determine the test tables and views that you need to test your application. Create a test table on your list when either of the following conditions exists:
 - The application modifies data in the table.
 - You need to create a view that is based on a test table because your application modifies data in the view.

To continue the example, create these test tables:

- TEST.EMP, with the following format:

EMPNO	LASTNAME	WORKDEPT	PHONENO	JOB
⋮	⋮	⋮	⋮	⋮

- TEST.PROJ, with the same columns and format as DSN8A10.PROJ, because the application inserts rows into the DSN8A10.PROJ table.

To support the example, create a test view of the DSN8A10.DEPT table.

- TEST.DEPT view, with the following format:

DEPTNO	MGRNO
⋮	⋮

Because the application does not change any data in the DSN8A10.DEPT table, you can base the view on the table itself (rather than on a test table). However, a safer approach is to have a complete set of test tables and to test the program thoroughly using only test data.

Authorization for test tables and applications

Before you can create a table, you need to be authorized to create tables and to use the table space in which the table is to reside. You must also have authority to bind and run programs that you want to test.

Your DBA can grant you the necessary authorization to create and access tables and to bind and run programs.

If you intend to use existing tables and views (either directly or as the basis for a view), you need privileges to access those tables and views. Your DBA can grant those privileges.

To create a view, you must have authorization for each table and view on which you base the view. You then have the same privileges over the view that you have over the tables and views on which you based the view. Before trying the examples, have your DBA grant you the privileges to create new tables and views and to access existing tables. Obtain the names of tables and views that you are authorized to access (as well as the privileges you have for each table) from your DBA.

Example SQL statements to create a comprehensive test structure

You need to create a storage group, database, table space, and table to use as a test structure for your SQL application.

The following SQL statements show how to create a complete test structure to contain a small table named SPUFINUM. The test structure consists of:

- A storage group named SPUFISG
- A database named SPUFIDB
- A table space named SPUFITS in database SPUFIDB and using storage group SPUFISG
- A table named SPUFINUM within the table space SPUFITS

```
CREATE STOGROUP SPUFISG
  VOLUMES (user-volume-number)
  VCAT DSNCAT ;
```

```
CREATE DATABASE SPUFIDB ;
```

```
CREATE TABLESPACE SPUFITS
  IN SPUFIDB
  USING STOGROUP SPUFISG ;
```

```
CREATE TABLE SPUFINUM
  ( XVAL CHAR(12) NOT NULL,
    ISFLOAT FLOAT,
    DEC30 DECIMAL(3,0),
    DEC31 DECIMAL(3,1),
    DEC32 DECIMAL(3,2),
    DEC33 DECIMAL(3,3),
    DEC10 DECIMAL(1,0),
    DEC11 DECIMAL(1,1),
```

```
DEC150 DECIMAL(15,0),
DEC151 DECIMAL(15,1),
DEC1515 DECIMAL(15,15) )
IN SPUFIDB.SPUFITS ;
```

Related reference:

- [➤ CREATE DATABASE \(DB2 SQL\)](#)
- [➤ CREATE STOGROUP \(DB2 SQL\)](#)
- [➤ CREATE TABLE \(DB2 SQL\)](#)
- [➤ CREATE TABLESPACE \(DB2 SQL\)](#)

Populating the test tables with data

To populate test tables, use SQL INSERT statements or the LOAD utility.

You can put test data into a table in several ways:

- INSERT ... VALUES (an SQL statement) puts one row into a table each time the statement executes.
- INSERT ... SELECT (an SQL statement) obtains data from an existing table (based on a SELECT clause) and puts it into the table that is identified in the INSERT statement.
- MERGE (an SQL statement) puts new data into a table and updates existing data.
- The LOAD utility obtains data from a sequential file (a non-DB2 file), formats it for a table, and puts it into a table.
- The DB2 sample UNLOAD program (DSNTIAUL) can unload data from a table or view and build control statements for the LOAD utility.
- The UNLOAD utility can unload data from a table and build control statements for the LOAD utility.

Related concepts:

“DB2 sample applications” on page 1092

Related tasks:

“Inserting rows by using the INSERT statement” on page 655

“Inserting rows into a table from another table” on page 657

“Inserting data and updating data in a single operation” on page 661

Related reference:

- [➤ LOAD \(DB2 Utilities\)](#)
- [➤ UNLOAD \(DB2 Utilities\)](#)

Methods for testing SQL statements

You can test your SQL statements by using SQL Processing Using File Input (SPUFI) or the command line processor.

Test with SPUFI: You can use SPUFI (an interface between ISPF and DB2) to test SQL statements in a TSO/ISPF environment. With SPUFI panels, you can put SQL statements into a data set that DB2 subsequently executes. The SPUFI Main panel has several functions that enable you to:

- Name an input data set to hold the SQL statements that are passed to DB2 for execution

- Name an output data set to contain the results of executing the SQL statements
- Specify SPUFI processing options

Test with the command line processor: You can use the command line processor to test SQL statements from UNIX System Services on z/OS.

SQL statements that are executed under SPUFI or the command line processor operate on actual tables (in this case, the tables that you created for testing). Consequently, before you access DB2 data:

- Make sure that all tables and views that your SQL statements refer to exist.
- If the tables or views do not exist, create them (or have your database administrator create them). You can use SPUFI or the command line processor to issue the CREATE statements that are used to create the tables and views that you need for testing.

Related concepts:

 [Command line processor \(DB2 Commands\)](#)

Related tasks:

[“Executing SQL by using SPUFI”](#)

Executing SQL by using SPUFI

You can execute SQL statements dynamically in a TSO session by using the SPUFI (SQL processor using file input) facility.

Before you use SPUFI, allocate an input data set to store the SQL statements that you want to execute, if such a data set does not already exist.

Before you begin this task, you can specify whether TSO message IDs are displayed by using the TSO PROFILE command. To view message IDs, type TSO PROFILE MSGID on the ISPF command line. To suppress message IDs, type TSO PROFILE NOMSGID.

These instructions assume that ISPF is available to you.

Important: Ensure that the TSO terminal CCSID matches the DB2 CCSID. If these CCSIDs do not match, data corruption can occur. If SPUFI issues the warning message DSNE345I, terminate your SPUFI session and notify the system administrator.

SPUFI can execute SQL statements that retrieve Unicode UTF-16 graphic data. However, SPUFI might not be able to display some characters, if those characters have no mapping in the target SBCS EBCDIC CCSID.

To execute SQL by using SPUFI:

1. Open SPUFI and specify the initial options.
2. **Optional:** “Changing SPUFI defaults” on page 1041
3. Enter SQL statements in SPUFI.
4. Process SQL statements with SPUFI.

Opening SPUFI and specifying initial options:

To begin using SPUFI, you need to open and fill out the SPUFI panel.

To open SPUFI and specify initial options:

1. Select SPUFI from the DB2I Primary Option Menu as shown in Figure 53 on page 981. The SPUFI panel is displayed.
2. Specify the input data set name and output data set name. An example of a SPUFI panel in which an input data set and output data set have been specified is shown in the following figure.

```
DSNESP01                SPUFI                SSID: DSN
===>
Enter the input data set name: (Can be sequential or partitioned)
 1 DATA SET NAME..... ==> EXAMPLES(XMP1)
 2 VOLUME SERIAL..... ==> (Enter if not cataloged)
 3 DATA SET PASSWORD. ==> (Enter if password protected)

Enter the output data set name: (Must be a sequential data set)
 4 DATA SET NAME..... ==> RESULT

Specify processing options:
 5 CHANGE DEFAULTS... ==> Y (Y/N - Display SPUFI defaults panel?)
 6 EDIT INPUT..... ==> Y (Y/N - Enter SQL statements?)
 7 EXECUTE..... ==> Y (Y/N - Execute SQL statements?)
 8 AUTOCOMMIT..... ==> Y (Y/N - Commit after successful run?)
 9 BROWSE OUTPUT..... ==> Y (Y/N - Browse output data set?)

For remote SQL processing:
10 CONNECT LOCATION ==>

PRESS: ENTER to process      END to exit      HELP for more information
```

Figure 75. The SPUFI panel filled in

3. **Optional:** Specify new values in any of the other fields on the SPUFI panel. For more information about these fields, see “The SPUFI panel” on page 1039.

Entering SQL statements in SPUFI:

After you open SPUFI, specify the initial options, and optionally change any SPUFI defaults, you can enter one or more SQL statements to execute.

Before you begin this task, you must complete the task "Opening SPUFI and specifying initial options."

If the input data set that you specified on the SPUFI panel already contains all of the SQL statements that you want to execute, you can bypass this editing step by specifying NO for the EDIT INPUT field on the SPUFI panel.

To enter SQL statements by using SPUFI:

1. If the EDIT panel is not already open, on the SPUFI panel, specify Y in the EDIT INPUT field and press ENTER. If the input data set that you specified is empty, an empty EDIT panel opens. Otherwise, if the input data set contained SQL statements, those SQL statements are displayed in an EDIT panel.
2. On the EDIT panel, use the ISPF EDIT program to enter or edit any SQL statements that you want to execute. Move the cursor to the first blank input line, and enter the first part of an SQL statement. You can enter the rest of the SQL statement on subsequent lines, as shown in the following figure:

```

EDIT -----userid.EXAMPLES(XMP1) ----- COLUMNS 001 072
COMMAND INPUT ==> SAVE                      SCROLL ==> PAGE
***** TOP OF DATA *****
000100 SELECT LASTNAME, FIRSTNME, PHONENO
000200   FROM DSN8A10.EMP
000300   WHERE WORKDEPT= 'D11'
000400   ORDER BY LASTNAME;
***** BOTTOM OF DATA *****

```

Figure 76. The edit panel: After entering an SQL statement

Consider the following rules and recommendations when editing this input data set:

- Indent your lines and enter your statements on several lines to make your statements easier to read. Entering your statements on multiple lines does not change how your statements are processed.
 - Do not put more than one SQL statement on a single line. If you do, the first statement executes, but DB2 ignores the other SQL statements on the same line. You can put more than one SQL statement in the input data set. DB2 executes the statements in the order in which you placed them in the data set.
 - End each SQL statement with the statement terminator that you specified on the CURRENT SPUFI DEFAULTS panel.
 - Save the data set every 10 minutes or so by entering the SAVE command.
3. Press the END PF key. The data set is saved, and the SPUFI panel is displayed.

Processing SQL statements with SPUFI:

You can use SPUFI to submit the SQL statements in a data set to DB2.

Before you begin this task, you must:

- Complete the task "Opening SPUFI and specifying initial options."
- Ensure that the input data set contains the SQL statements that you want to execute.

To process SQL statements by using SPUFI:

1. On the SPUFI panel, specify YES in the EXECUTE field.
2. If you did not just finish using the EDIT panel to edit the input data set as described in "Entering SQL statements in SPUFI," specify NO In the EDIT INPUT field.
3. Press Enter.

SPUFI passes the input data set to DB2 for processing. DB2 executes the SQL statement in the input data set and sends the output to the output data set.

The output data set opens.

Your SQL statement might take a long time to execute, depending on how large a table DB2 must search, or on how many rows DB2 must process. In this case, you can interrupt the processing by pressing the PA1 key. Then respond to the message that asks you if you really want to stop processing. This action cancels the executing SQL statement. Depending on how much of the input data set DB2 was able to process before you interrupted its processing, DB2 might not have opened the output data set yet, or the output data set might contain all or part of the results data that are produced so far.

For information about how to interpret the output in the output data set, see “Output from SPUFI” on page 1048.

SQL statements that exceed resource limit thresholds:

Your system administrator might use the DB2 resource limit facility (governor) to set time limits for processing SQL statements in SPUFI. Those limits can be error limits or warning limits.

If you execute an SQL statement through SPUFI that runs longer than this error time limit, SPUFI terminates processing of that SQL statement and all statements that follow in the SPUFI input data set. SPUFI displays a panel that lets you commit or roll back the previously uncommitted changes that you have made. That panel is shown in the following figure.

```
DSNESP04          SQL STATEMENT RESOURCE LIMIT EXCEEDED          SSID: DSN
===>

The following SQL statement has encountered an SQLCODE of -905 or -495:

Statement text

Your SQL statement has exceeded the resource utilization threshold set
by your site administrator.

You must ROLLBACK or COMMIT all the changes made since the last COMMIT.
SPUFI processing for the current input file will terminate immediately
after the COMMIT or ROLLBACK is executed.

1 NEXT ACTION ===>          (Enter COMMIT or ROLLBACK)

PRESS:  ENTER to process          HELP for more information
```

Figure 77. The resource limit facility error panel

If you execute an SQL statement through SPUFI that runs longer than the warning time limit for predictive governing, SPUFI displays the SQL STATEMENT RESOURCE LIMIT EXCEEDED panel. On this panel, you can tell DB2 to continue executing that statement, or stop processing that statement and continue to the next statement in the SPUFI input data set. That panel is shown in the following figure.


```

DSNESP05          SQL STATEMENT RESOURCE LIMIT EXCEEDED          SSID: DSN
===>

The following SQL statement has encountered an SQLCODE of 495:

Statement text

You can now either CONTINUE executing this statement or BYPASS the execution
of this statement. SPUFI processing for the current input file will continue
after the CONTINUE or BYPASS processing is completed.

1 NEXT ACTION ===>          (Enter CONTINUE or BYPASS)

PRESS:  ENTER to process          HELP for more information

```

Figure 78. The resource limit facility warning panel

Related tasks:

[➤](#) Setting limits for system resource usage by using the resource limit facility (DB2 Performance)

Related reference:

[➤](#) ISPF User's Guide Vol II (z/OS V1R7.0 ISPF User's Guide Vol II)

Content of a SPUFI input data set

A SPUFI input data set can contain SQL statements, comments, and SPUFI control statements.

You can put comments about SQL statements either on separate lines or on the same line. In either case, use two hyphens (--) to begin a comment. Specify any text other than #SET TERMINATOR or #SET TOLWARN after the comment marker. DB2 ignores everything else to the right of the two hyphens.

The SPUFI panel

The SPUFI panel is the first panel that you need to fill out to run the SPUFI application.

After you complete any fields on the SPUFI panel and press Enter, those settings are saved. When the SPUFI panel displays again, the data entry fields on the panel contain the values that you previously entered. You can specify data set names and processing options each time the SPUFI panel is displayed, as needed. Values that you do not change remain in effect.

The following descriptions explain the fields that are available on the SPUFI panel.

1,2,3 INPUT DATA SET NAME

Identify the input data set in fields 1 through 3. This data set contains one or more SQL statements that you want to execute. Allocate this data set before you use SPUFI, if one does not already exist. Consider the following rules:

- The name of the data set must conform to standard TSO naming conventions.

- The data set can be empty before you begin the session. You can then add the SQL statements by editing the data set from SPUFI.
- The data set can be either sequential or partitioned, but it must have the following DCB characteristics:
 - A record format (RECFM) of either F or FB.
 - A logical record length (LRECL) of either 79 or 80. Use 80 for any data set that the EXPORT command of DB2 QMF did not create.
- Data in the data set can begin in column 1. It can extend to column 71 if the logical record length is 79, and to column 72 if the logical record length is 80. SPUFI assumes that the last 8 bytes of each record are for sequence numbers.

If you use this panel a second time, the name of the data set you previously used displays in the field DATA SET NAME. To create a new member of an existing partitioned data set, change only the member name.

4 OUTPUT DATA SET NAME

Enter the name of a data set to receive the output of the SQL statement. You do not need to allocate the data set before you do this.

If the data set exists, the new output replaces its content. If the data set does not exist, DB2 allocates a data set on the device type specified on the CURRENT SPUFI DEFAULTS panel and then catalogs the new data set. The device must be a direct-access storage device, and you must be authorized to allocate space on that device.

Attributes required for the output data set are:

- Organization: sequential
- Record format: F, FB, FBA, V, VB, or VBA
- Record length: 80 to 32768 bytes, not less than the input data set

“Executing SQL by using SPUFI” on page 1035 shows the simplest choice, entering **RESULT**. SPUFI allocates a data set named *userid.RESULT* and sends all output to that data set. If a data set named *userid.RESULT* already exists, SPUFI sends DB2 output to it, replacing all existing data.

5 CHANGE DEFAULTS

Enables you to change control values and characteristics of the output data set and format of your SPUFI session. If you specify **Y(YES)** you can look at the SPUFI defaults panel. See “Changing SPUFI defaults” on page 1041 for more information about the values you can specify and how they affect SPUFI processing and output characteristics. You do not need to change the SPUFI defaults for this example.

6 EDIT INPUT

To edit the input data set, leave **Y(YES)** on line 6. You can use the ISPF editor to create a new member of the input data set and enter SQL statements in it. (To process a data set that already contains a set of SQL statements you want to execute immediately, enter **N (NO)**. Specifying **N** bypasses the step 3 described in “Executing SQL by using SPUFI” on page 1035.)

7 EXECUTE

To execute SQL statements contained in the input data set, leave **Y(YES)** on line 7.

SPUFI handles the SQL statements that can be dynamically prepared.

8 AUTOCOMMIT

To make changes to the DB2 data permanent, leave **Y(YES)** on line 8.

Specifying **Y** makes SPUIFI issue COMMIT if all statements execute successfully. If all statements do not execute successfully, SPUIFI issues a ROLLBACK statement, which deletes changes already made to the file (back to the last commit point).

If you specify **N**, DB2 displays the SPUIFI COMMIT OR ROLLBACK panel after it executes the SQL in your input data set. That panel prompts you to COMMIT, ROLLBACK, or DEFER any updates made by the SQL. If you enter DEFER, you neither commit nor roll back your changes.

9 BROWSE OUTPUT




To look at the results of your query, leave **Y(YES)** on line 9. SPUIFI saves the results in the output data set. You can look at them at any time, until you delete or write over the data set.

10 CONNECT LOCATION

Specify the name of the database server, if applicable, to which you want to submit SQL statements. SPUIFI then issues a type 2 CONNECT statement to this server.

SPUIFI is a locally bound package. SQL statements in the input data set can process only if the CONNECT statement is successful. If the connect request fails, the output data set contains the resulting SQL return codes and error messages.

Related reference:

-  Characteristics of SQL statements in DB2 for z/OS (DB2 SQL)
-  COMMIT (DB2 SQL)
-  ROLLBACK (DB2 SQL)

Changing SPUIFI defaults

Before you execute SQL statements in SPUIFI, you can change the default execution behavior, such as the SQL terminator and the isolation level.

SPUIFI provides default values the first time that you use SPUIFI for all options except the DB2 subsystem name. Any changes that you make to these values remain in effect until you change the values again.

To change the SPUIFI defaults:

1. On the SPUIFI panel, specify **YES** in the CHANGE DEFAULTS field.
2. Press Enter. The CURRENT SPUIFI DEFAULTS panel opens. The following figure shows the initial default values.

```

DSNESP02                CURRENT SPUFI DEFAULTS                SSID: DSN
====>
Enter the following to control your SPUFI session:
 1 SQL TERMINATOR .. ==> ;      (SQL Statement Terminator)
 2 ISOLATION LEVEL  ==> RR      (RR=Repeatable Read, CS=Cursor Stability)
                                UR=Uncommitted Read)
 3 MAX SELECT LINES ==> 250     (Maximum lines to be returned from a SELECT)
 4 ALLOW SQL WARNINGS==> NO     (Continue fetching after SQL warning)
 5 CHANGE PLAN NAMES ==> NO     (Change the plan names used by SPUFI)
 6 SQL FORMAT ..... ==> SQL     (SQL, SQLCOMNT, or SQLPL)
Output data set characteristics:
 7 SPACE UNIT ..... ==> TRK     (TRK or CYL)
 8 PRIMARY SPACE ... ==> 5      (Primary space allocation 1-999)
 9 SECONDARY SPACE . ==> 6      (Secondary space allocation 0-999)
10 RECORD LENGTH ... ==> 4092   (LRECL= logical record length)
11 BLOCKSIZE ..... ==> 4096    (Size of one block)
12 RECORD FORMAT.... ==> VB     (RECFM= F, FB, FBA, V, VB, or VB)
13 DEVICE TYPE..... ==> SYSDA   (Must be a DASD unit name)
Output format characteristics:
14 MAX NUMERIC FIELD ==> 33     (Maximum width for numeric field)
15 MAX CHAR FIELD .. ==> 80     (Maximum width for character field)
16 COLUMN HEADING .. ==> NAMES  (NAMES, LABELS, ANY, or BOTH)

PRESS: ENTER to process      END to exit      HELP for more information

```

Figure 79. The SPUFI defaults panel

3. Specify any new values in the fields of this panel. All fields must contain a value.
4. Press Enter. SPUFI saves your changes and one of the following panels or data sets open:
 - The CURRENT SPUFI DEFAULTS - PANEL 2 panel. This panel opens if you specified YES in the CHANGE PLAN NAMES field.
 - EDIT panel. This panel opens if you specified YES in the EDIT INPUT field on the SPUFI panel.
 - Output data set. This data set opens if you specified NO in the EDIT INPUT field on the SPUFI panel.
 - SPUFI panel. This panel opens if you specified NO for all of the processing options on the SPUFI panel.

If you press the END key on the CURRENT SPUFI DEFAULTS panel, the SPUFI panel is displayed, and you lose all the changes that you made on the CURRENT SPUFI DEFAULTS panel.

5. If the CURRENT SPUFI DEFAULTS - PANEL 2 panel opens, specify values for the fields on that panel and press Enter. All fields must contain a value.

Important: If you specify an invalid or incorrect plan name, SPUFI might experience operational errors or your data might be contaminated.

SPUFI saves your changes and one of the following panels or data sets open:

- EDIT panel. This panel opens if you specified YES in the EDIT INPUT field on the SPUFI panel.
- Output data set. This data set opens if you specified NO in the EDIT INPUT field on the SPUFI panel.
- SPUFI panel. This panel opens if you specified NO for all of the processing options on the SPUFI panel.

Next, continue with one of the following tasks:

- If you want to add SQL statements to the input data set or edit the SQL statements in the input data set, enter SQL statements in SPUFI.
- Otherwise if the input data set already contains the SQL statements that you want to execute, process SQL statements with SPUFI.

Related reference:

“CURRENT SPUFI DEFAULTS panel”

“CURRENT SPUFI DEFAULTS - PANEL 2 panel” on page 1046

CURRENT SPUFI DEFAULTS panel

Use the CURRENT SPUFI DEFAULTS panel to specify SPUFI default values.

The following descriptions explain the information on the CURRENT SPUFI DEFAULTS panel.

1 SQL TERMINATOR

Specify the character that you use to end each SQL statement. You can specify any character *except* the characters listed in the following table. A semicolon (;) is the default SQL terminator.

Table 164. Invalid special characters for the SQL terminator

Name	Character	Hexadecimal representation
blank		X'40'
comma	,	X'5E'
double quote	"	X'7F'
left parenthesis	(X'4D'
right parenthesis)	X'5D'
single quote	'	X'7D'
underscore	_	X'6D'

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose you choose the character # as the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like the following statement:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

A CREATE PROCEDURE statement with embedded semicolons looks like the following statement:

```
CREATE PROCEDURE PROC1 (IN PARM1 INT, OUT SCODE INT)
  LANGUAGE SQL
  BEGIN
    DECLARE SQLCODE INT;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
      SET SCODE = SQLCODE;
    UPDATE TBL1 SET COL1 = PARM1;
  END #
```

Be careful to choose a character for the SQL terminator that is not used within the statement.

You can also set or change the SQL terminator within a SPUFI input data set by using the `--#SET TERMINATOR` statement.

2 ISOLATION LEVEL

Specify the isolation level for your SQL statements.

3 MAX SELECT LINES

The maximum number of rows that a SELECT statement can return. To limit the number of rows retrieved, enter another maximum number greater than 1.

4 ALLOW SQL WARNINGS

Enter YES or NO to indicate whether SPUFI will continue to process an SQL statement after receiving SQL warnings:

YES If a warning occurs when SPUFI executes an OPEN or FETCH for a SELECT statement, SPUFI continues to process the SELECT statement.

NO If a warning occurs when SPUFI executes an OPEN or FETCH for a SELECT statement, SPUFI stops processing the SELECT statement. If SQLCODE +802 occurs when SPUFI executes a FETCH for a SELECT statement, SPUFI continues to process the SELECT statement.

You can also specify how SPUFI pre-processes the SQL input by using the `--#SET TOLWARN` statement.

5 CHANGE PLAN NAMES

If you enter YES in this field, you can change plan names on a subsequent SPUFI defaults panel, DSNESP07. Enter YES in this field only if you are certain that you want to change the plan names that are used by SPUFI. Consult with your DB2 system administrator if you are uncertain whether you want to change the plan names. Using an invalid or incorrect plan name might cause SPUFI to experience operational errors or it might cause data contamination.

6 SQL FORMAT

Specify how SPUFI pre-processes the SQL input before passing it to DB2. Select one of the following options:

SQL This is the preferred mode for SQL statements other than SQL procedural language. When you use this option, which is the default, SPUFI collapses each line of an SQL statement into a single line before passing the statement to DB2. SPUFI also discards all SQL comments.

SQLCOMNT

This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, behavior is similar to SQL mode, except that SPUFI does not discard SQL comments. Instead, it automatically terminates each SQL comment with a line feed character (hex 25), unless the comment is already terminated by one or more line formatting characters. Use this option to process SQL procedural language with minimal modification by SPUFI.

SQLPL

This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, SPUFI retains SQL comments and terminates each line of an SQL

statement with a line feed character (hex 25) before passing the statement to DB2. Lines that end with a split token are not terminated with a line feed character. Use this mode to obtain improved diagnostics and debugging of SQL procedural language.

You can also specify how SPUFI pre-processes the SQL input by using the `--#SET SQLFORMAT` statement.

7 SPACE UNIT

Specify how space for the SPUFI output data set is to be allocated.

TRK Track

CYL Cylinder

8 PRIMARY SPACE

Specify how many tracks or cylinders of primary space are to be allocated.

9 SECONDARY SPACE

Specify how many tracks or cylinders of secondary space are to be allocated.

10 RECORD LENGTH

The record length must be at least 80 bytes. The maximum record length depends on the device type you use. The default value allows a 32756-byte record.

Each record can hold a single line of output. If a line is longer than a record, the output is truncated, and SPUFI discards fields that extend beyond the record length.

11 BLOCKSIZE

Follow the normal rules for selecting the block size. For record format F, the block size is equal to the record length. For FB and FBA, choose a block size that is an even multiple of LRECL. For VB and VBA only, the block size must be 4 bytes larger than the block size for FB or FBA.

12 RECORD FORMAT

Specify F, FB, FBA, V, VB, or VBA. FBA and VBA formats insert a printer control character after the number of lines specified in the LINES/PAGE OF LISTING field on the DB2I Defaults panel. The record format default is VB (variable-length blocked).

13 DEVICE TYPE

Specify a standard z/OS name for direct-access storage device types. The default is SYSDA. SYSDA specifies that z/OS is to select an appropriate direct access storage device.

14 MAX NUMERIC FIELD

The maximum width of a numeric value column in your output. Choose a value greater than 0. The default is 33.

15 MAX CHAR FIELD

The maximum width of a character value column in your output. DATETIME and GRAPHIC data strings are externally represented as characters, and SPUFI includes their defaults with the default values for character fields. Choose a value greater than 0. The IBM-supplied default is 250.

16 COLUMN HEADING

You can specify NAMES, LABELS, ANY, or BOTH for column headings.

- NAMES uses column names only.

- LABELS (default) uses column labels. Leave the title blank if no label exists.
- ANY uses existing column labels or column names.
- BOTH creates two title lines, one with names and one with labels.

Column names are the column identifiers that you can use in SQL statements. If an SQL statement has an AS clause for a column, SPUIFI displays the contents of the AS clause in the heading, rather than the column name. You define column labels with LABEL statements.

Related concepts:

“Output from SPUIFI” on page 1048

Related tasks:

“Changing SPUIFI defaults” on page 1041

“Executing SQL by using SPUIFI” on page 1035

CURRENT SPUIFI DEFAULTS - PANEL 2 panel

Use the CURRENT SPUIFI DEFAULTS - PANEL 2 panel to specify default plan name information.

This panel opens if you specify YES in the CHANGE PLAN NAMES field of the CURRENT SPUIFI DEFAULTS panel.

Figure 80 shows the initial default values.

```

DSNESP07                CURRENT SPUIFI DEFAULTS - PANEL 2                SSID: DSN
====>
Enter the following to control your SPUIFI session:
 1 CS ISOLATION PLAN    ==>> DSNESPCS (Name of plan for CS isolation level)
 2 RR ISOLATION PLAN    ==>> DSNESPRR (Name of plan for RR isolation level)
 3 UR ISOLATION PLAN    ==>> DSNESPUR (Name of plan for UR isolation level)

Indicate warning message status:
 4 BLANK CCSID WARNING ==>> YES      (Show warning if terminal CCSID is blank)

PRESS: ENTER to process   END to exit   HELP for more information

```

Figure 80. CURRENT SPUIFI DEFAULTS - PANEL 2

The following descriptions explain the information on the CURRENT SPUIFI DEFAULTS - PANEL 2 panel.

1 CS ISOLATION PLAN

Specify the name of the plan that SPUIFI uses when you specify an isolation level of cursor stability (CS). By default, this name is DSNESPCS.

2 RR ISOLATION PLAN

Specify the name of the plan that SPUIFI uses when you specify an isolation level of repeatable read (RR). By default, this name is DSNESPRR.

3 UR ISOLATION PLAN

Specify the name of the plan that SPUIFI uses when you specify an isolation level of uncommitted read (UR). By default, this name is DSNESPUR.

4 BLANK CCSID ALERT

Indicate whether to receive message DSNE345I when the terminal CCSID setting is blank. A blank terminal CCSID setting occurs when the terminal code page and character set cannot be queried or if they are not supported by ISPF.

Recommendation: To avoid possible data contamination use the default setting of YES, unless you are specifically directed by your DB2 system administrator to use NO.

Setting the SQL terminator character in a SPUIFI input data set

In the SPUIFI input data set, you can override the SQL terminator character that is specified on the CURRENT SPUIFI DEFAULTS panel. The default SQL terminator is a semicolon (;).

Overriding the default SQL termination character is useful if you need to use a different SQL terminator character for one particular SQL statement.

To set the SQL terminator character in a SPUIFI input data set, specify the text `--#SET TERMINATOR character` before that SQL statement to which you want this character to apply. This text specifies that SPUIFI is to interpret *character* as a statement terminator. You can specify any single-byte character **except** the characters that are listed in Table 165. Choose a character for the SQL terminator that is not used within the statement. The terminator that you specify overrides a terminator that you specified in option 1 of the CURRENT SPUIFI DEFAULTS panel or in a previous `--#SET TERMINATOR` statement.

Table 165. Invalid special characters for the SQL terminator

Name	Character	Hexadecimal representation
blank		X'40'
comma	,	X'5E'
double quote	"	X'7F'
left parenthesis	(X'4D'
right parenthesis)	X'5D'
single quote	'	X'7D'
underscore	_	X'6D'

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose that you choose the character # as the statement terminator. In this case, a CREATE TRIGGER statement with embedded semicolons looks like this:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

Controlling toleration of warnings in SPUFI

When you use SPUFI, you can specify the action that SPUFI is to take when a warning occurs.

To control the toleration of warnings, specify one of the following TOLWARN control statements:

--#SET TOLWARN NO

If a warning occurs when SPUFI executes an OPEN or FETCH for SELECT statement, SPUFI stops processing the SELECT statement. If SQLCODE +802 occurs when SPUFI executes a FETCH for a SELECT statement, SPUFI continues to process the SELECT statement.

--#SET TOLWARN YES

If a warning occurs when SPUFI executes an OPEN or FETCH for SELECT statement, SPUFI continues to process the SELECT statement.

The following example activates and then deactivates toleration of SQL warnings:

```
SELECT * FROM MY.T1;  
--#SET TOLWARN YES  
SELECT * FROM YOUR.T1;  
--#SET TOLWARN NO
```

Output from SPUFI

SPUFI formats and displays the output data set using the ISPF Browse program.

Figure 81 on page 1049 shows the output from the sample program. An output data set contains the following items for each SQL statement that DB2 executes:

- The executed SQL statement, copied from the input data set
- The results of executing the SQL statement
- The formatted SQLCA, if an error occurs during statement execution

At the end of the data set are summary statistics that describe the processing of the input data set as a whole.

For SELECT statements that are executed with SPUFI, the message "SQLCODE IS 100" indicates an error-free result. If the message SQLCODE IS 100 is the only result, DB2 is unable to find any rows that satisfy the condition that is specified in the statement.

For all other types of SQL statements that are executed with SPUFI, the message "SQLCODE IS 0" indicates an error-free result.

```

BROWSE-- userid.RESULT                                COLUMNS 001 072
COMMAND INPUT ==>>>                                SCROLL ==>> PAGE
-----+-----+-----+-----+-----+-----+-----+
SELECT LASTNAME, FIRSTNME, PHONENO                    00010000
FROM DSN8A10.EMP                                     00020000
WHERE WORKDEPT = 'D11'                               00030000
ORDER BY LASTNAME;                                   00040000
-----+-----+-----+-----+-----+
LASTNAME      FIRSTNME      PHONENO
ADAMSON       BRUCE          4510
BROWN         DAVID          4501
JOHN          REBA           0672
JONES         WILLIAM        0942
LUTZ          JENNIFER       0672
PIANKA        ELIZABETH      3782
SCOUTTEN      MARILYN        1682
STERN         IRVING         6423
WALKER        JAMES          2986
YAMAMOTO      KIYOSHI        2890
YOSHIMURA    MASATOSHI      2890
DSNE610I NUMBER OF ROWS DISPLAYED IS 11
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
-----+-----+-----+-----+-----+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
-----+-----+-----+-----+-----+
DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
DSNE621I NUMBER OF INPUT RECORDS READ IS 4
DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 30

```

Figure 81. Result data set from the sample problem

Formatting rules for SELECT statement results in SPUFI:

The results of SELECT statements follow these rules:

- If numeric or character data of a column cannot be displayed completely:
 - Character values and binary values that are too wide truncate on the right.
 - Numeric values that are too wide display as asterisks (*).
 - For columns other than LOB and XML columns, if truncation occurs, the output data set contains a warning message. Because LOB and XML columns are generally longer than the value you choose for field MAX CHAR FIELD on panel CURRENT SPUFI DEFAULTS, SPUFI displays no warning message when it truncates LOB or XML column output.

You can change the amount of data that is displayed for numeric and character columns by changing values on the CURRENT SPUFI DEFAULTS panel, as described in “Changing SPUFI defaults” on page 1041.

- A null value is displayed as a series of hyphens (-).
- A ROWID, BLOB, BINARY, or VARBINARY column value is displayed in hexadecimal.
- A CLOB column value is displayed in the same way as a VARCHAR column value.
- A DBCLOB column value is displayed in the same way as a VARGRAPHIC column value.
- An XML column is displayed in the same way as a LOB column.
- A heading identifies each selected column, and is repeated at the top of each output page. The contents of the heading depend on the value that you specified in the COLUMN HEADING field of the CURRENT SPUFI DEFAULTS panel.

Content of the messages from SPUFI:

Each SPUFI message contains the following:

- The SQLCODE, if the statement executes successfully.
- The formatted SQLCA, if the statement executes unsuccessfully.
- What character positions of the input data set that SPUFI scanned to find SQL statements. This information helps you check the assumptions that SPUFI made about the location of line numbers (if any) in your input data set.
- Some overall statistics:
 - Number of SQL statements that are processed
 - Number of input records that are read (from the input data set)
 - Number of output records that are written (to the output data set).

Other messages that you could receive from the processing of SQL statements include:

- The number of rows that DB2 processed, that either:
 - Your select operation retrieved
 - Your update operation modified
 - Your insert operation added to a table
 - Your delete operation deleted from a table
- Which columns display truncated data because the data was too wide

Testing an external user-defined function

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where user-defined functions run. You need to use alternative testing strategies.

Testing a user-defined function by using the Debug Tool for z/OS

You can use the Debug Tool for z/OS to test DB2 for z/OS user-defined functions that are written in any of the supported languages. The Debug Tool for z/OS works with Language Environment.

You can use the Debug Tool either interactively or in batch mode. To test your user-defined function using the Debug Tool, you must have the Debug Tool installed on the z/OS system where the user-defined function runs.

To test a user-defined function by using the Debug Tool for z/OS, choose one of the following approaches:

- To use the Debug Tool interactively:
 1. Compile the user-defined function with the TEST option. This places information in the program that the Debug Tool uses.
 2. Invoke the Debug Tool. One way to do that is to specify the Language Environment run time TEST option. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run time options is with the RUN OPTIONS clause of CREATE FUNCTION or ALTER FUNCTION. For example, suppose that you code this option:

```
TEST(ALL,*,PROMPT,JBONES%SESSNA:)
```

The parameter values cause the following things to happen:

ALL

The Debug Tool gains control when an attention interrupt,abend, or program or Language Environment condition of Severity 1 and above occurs.

- * Debug commands will be entered from the terminal.

PROMPT

The Debug Tool is invoked immediately after Language Environment initialization.

JBONES%SESSNA:

The Debug Tool initiates a session on a workstation identified to APPC as JBJONES with a session ID of SESSNA.

3. If you want to save the output from your debugging session, issue a command that names a log file. For example, the following command starts logging to a file on the workstation called dbgtool.log.

```
SET LOG ON FILE dbgtool.log;
```

This should be the first command that you enter from the terminal or include in your commands file.

- To use the Debug Tool in batch mode:

1. If you plan to use the Language Environment run time TEST option to invoke the Debug Tool, compile the user-defined function with the TEST option. This places information in the program that the Debug Tool uses during a debugging session.
2. Allocate a log data set to receive the output from the Debug Tool. Put a DD statement for the log data set in the startup procedure for the stored procedures address space.
3. Enter commands in a data set that you want the Debug Tool to execute. Put a DD statement for that data set in the startup procedure for the stored procedures address space. To define the data set that contains the commands to the Debug Tool, specify its data set name or DD name in the TEST run time option. For example, this option tells the Debug Tool to look for the commands in the data set that is associated with DD name TESTDD:

```
TEST(ALL,TESTDD,PROMPT,*)
```

The first command in the commands data set should be:

```
SET LOG ON FILE ddname;
```

This command directs output from your debugging session to the log data set you defined in step 2. For example, if you defined a log data set with DD name INSPLOG in the start-up procedure for the stored procedures address space, the first command should be:

```
SET LOG ON FILE INSPLOG;
```

4. Invoke the Debug Tool. The following are two possible methods for invoking the Debug Tool:
 - Specify the Language Environment run time TEST option. The most convenient place to do that is in the RUN OPTIONS parameter of CREATE FUNCTION or ALTER FUNCTION.
 - Put CEETEST calls in the user-defined function source code. If you use this approach for an existing user-defined function, you must compile, link-edit, and bind the user-defined function again. Then you must issue the STOP FUNCTION SPECIFIC and START FUNCTION SPECIFIC commands to reload the user-defined function.

You can combine the Language Environment run time TEST option with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

You can combine the Language Environment run time TEST option with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

Related reference:

“Components of a user-defined function definition” on page 523

 [Debug Tool for z/OS](#)

Testing a user-defined function by routing the debugging messages to SYSPRINT

You can include simple print statements in your user-defined function code that you route to SYSPRINT. Then use System Display and Search Facility (SDSF) to examine the SYSPRINT contents while the WLM-established stored procedure address space is running.

You can serialize I/O by running the WLM-established stored procedure address space with NUMTCB=1.

Testing a user-defined function by using driver applications

You can write a small driver application that calls a user-defined function as a subprogram and passes the parameter list for the user-defined function. You can then test and debug the user-defined function as a normal DB2 application under TSO.

You can then use TSO TEST and other commonly used debugging tools.

Testing a user-defined function by using SQL INSERT statements

You can use SQL to insert debugging information into a DB2 table. This allows other machines in the network (such as workstations) to easily access the data in the table by using DRDA access.

DB2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.

Debugging stored procedures

When debugging stored procedures, you might need to use different techniques than you would use for regular application programs. For example, some commonly used debugging tools, such as TSO TEST, are not available in the environment where stored procedures run.

To debug a stored procedure, perform one or more of the following actions:

- Take one or more of the following general actions, which are appropriate in many situations with stored procedures:
 - Ensure that all stored procedures are written to handle any SQL errors.

- Debug stored procedures as stand-alone programs on a workstation.
If you have debugging tools on a workstation, consider doing most of your development and testing on a workstation before installing a stored procedure on z/OS. This technique results in very little debugging activity on z/OS.
- Record stored procedure debugging messages to a disk file or JES spool file.
- Store debugging information in a table. This technique is especially useful for remote stored procedures.
- Use the DISPLAY command to view information about particular stored procedures, including statistics and thread information.
- In the stored procedure that you are debugging, issue DISPLAY commands. You can view the DISPLAY results in the SDSF output. The DISPLAY results can help you find information about the started task that is associated with the address space for the WLM application environment.
- If necessary, use the STOP PROCEDURE command to stop calls to one or more problematic stored procedures. You can restart them later.
- If your stored procedures address space has the CEEDUMP data set allocated, look at the diagnostic information in the CEEDUMP output.
- For COBOL, C, and C++ stored procedures, use the Debug Tool for z/OS.
- For COBOL stored procedures, compile the stored procedure with the option TEST(SYM) if you want a formatted local variable dump to be included in the CEEDUMP output.
- For native SQL procedures, external SQL procedures, and Java stored procedures, use the Unified Debugger.
- For external stored procedures, consider taking one or both of the following actions:
 - Use a driver application.
 - Create or alter the stored procedure definition to include the PARAMETER STYLE SQL option. This option enables the stored procedure to share any error information with the calling application. Ensure that your procedure follows linkage conventions for stored procedures.
- If you changed a stored procedure or a startup JCL procedure for a WLM application environment, determine whether you need to refresh the WLM environment. You must refresh the WLM environment before certain stored procedure changes take effect.

Related tasks:

“Handling SQL conditions in an SQL procedure” on page 578

 Displaying information about stored procedures with DB2 commands (DB2 Administration Guide)

 Refreshing WLM application environments for stored procedures (DB2 Administration Guide)


 Implementing DB2 stored procedures (DB2 Administration Guide)

Related reference:

“Linkage conventions for external stored procedures” on page 619

 -START PROCEDURE (DB2) (DB2 Commands)

 -STOP PROCEDURE (DB2) (DB2 Commands)

 Debugging COBOL, PL/I, and C/C++ procedures on z/OS (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

 Debugging SQL procedures on z/OS, Linux, UNIX, and Windows (DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond)

Debugging stored procedures with the Debug Tool and IBM VisualAge COBOL

If you have VisualAge® COBOL installed on your workstation and the Debug Tool installed on your z/OS system, you can use the VisualAge COBOL Edit/Compile/Debug component with the Debug Tool to debug COBOL stored procedures that run in a WLM-established stored procedures address space.

Before you begin debugging, write your COBOL stored procedure and set up the WLM environment.

To debug with the Debug Tool and IBM VisualAge COBOL:

1. When you compile the stored procedure, specify the TEST and SOURCE options. Ensure that the source listing is stored in a permanent data set. VisualAge COBOL displays the source listing during the debug session.
2. When you define the stored procedure, include run time option TEST with the suboption VADTCPIP&*ipaddr* in your RUN OPTIONS argument. VADTCPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge COBOL and is configured for TCP/IP communication with your z/OS system. *ipaddr* is the IP address of the workstation on which you display your debug information. For example, the RUN OPTIONS value in the following stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

```
CREATE PROCEDURE WLMCOB
  (IN INTEGER, INOUT VARCHAR(3000), INOUT INTEGER)
  MODIFIES SQL DATA
  LANGUAGE COBOL EXTERNAL
  PROGRAM TYPE MAIN
  WLM ENVIRONMENT WLMENV1
  RUN OPTIONS 'POSIX(ON),TEST(,,VADTCPIP&9.63.51.17:*)'
```
3. In the JCL startup procedure for WLM-established stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

```
//DSNWLM  PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=DSNA10.RUNLIB.LOAD
//        DD DISP=SHR,DSN=CEE.SCEERUN
//        DD DISP=SHR,DSN=DSNA10.SDSNLOAD
//        DD DISP=SHR,DSN=EQAW.SEQAMOD <== DEBUG TOOL
```
4. On the workstation, start the VisualAge Remote Debugger daemon. This daemon waits for incoming requests from TCP/IP.
5. Call the stored procedure. When the stored procedure starts, a window that contains the debug session is displayed on the workstation. You can then execute Debug Tool commands to debug the stored procedure.

Related reference:

 Debug Tool for z/OS

Debugging a C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS

You can debug a C or C++ stored procedure that runs in a WLM-established stored procedures address space. You must have the C/C++ Productivity Tools for z/OS installed on your workstation and the Debug Tool installed on your z/OS system.

The code against which you run the debug tools is the C source program that is produced by the program preparation process for the stored procedure.

Before you begin debugging, write your C++ stored procedure and set up the WLM environment.

To test the stored procedure with the Distributed Debugger feature of the C/C++ Productivity Tools for z/OS and the Debug Tool:

1. When you define the stored procedure, include run time option TEST with the suboption VADTCPIP&*ipaddr* in your RUN OPTIONS argument.
VADTCPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge C++ and is configured for TCP/IP communication with your z/OS system. *ipaddr* is the IP address of the workstation on which you display your debug information. For example, this RUN OPTIONS value in a stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

```
RUN OPTIONS 'POSIX(ON),TEST(,,VADTCPIP&9.63.51.17:*)'
```
2. Precompile the stored procedure. Ensure that the modified source program that is the output from the precompile step is in a permanent, catalogued data set.
3. Compile the output from the precompile step. Specify the TEST, SOURCE, and OPT(0) compiler options.
4. In the JCL startup procedure for the stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

```
//DSNWLM PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
// PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB DD DISP=SHR,DSN=DSNA10.RUNLIB.LOAD
// DD DISP=SHR,DSN=CEE.SCEERUN
// DD DISP=SHR,DSN=DSNA10.SDSNLOAD
// DD DISP=SHR,DSN=EQAW.SEQAMOD <== DEBUG TOOL
```
5. On the workstation, start the Distributed Debugger daemon. This daemon waits for incoming requests from TCP/IP.
6. Call the stored procedure. When the stored procedure starts, a window that contains the debug session is displayed on the workstation. You can then execute Debug Tool commands to debug the stored procedure.

Related reference:

 [Debug Tool for z/OS](#)

Debugging stored procedures by using the Unified Debugger

You can use the Unified Debugger to remotely debug native SQL procedures, external SQL procedures, and Java stored procedures that execute on DB2 for z/OS servers. The Unified Debugger also supports debugging nested stored procedure calls.


With the Unified Debugger, you can observe the execution of the procedure code, set breakpoints for lines, and view or modify variable values.

To debug stored procedures by using the Unified Debugger:

1. Set up the Unified Debugger by performing the following steps:
 - a. Ensure that job DSNTIJRT successfully created the stored procedures that provide server support for the Unified Debugger. This job is run during the installation and migration process. The stored procedures that this job creates must run in WLM environments.

Recommendation: Initially, define and use the DB2 core WLM environment DSNWLM_GENERAL to run the SYSPROC.DBG_RUNSESSIONMANAGER stored procedure and core WLM environment DSNWLM_DEBUGGER to run the other stored procedures for Unified debugger.
 - b. Define the debug mode characteristics for the stored procedure that you want to debug by completing one of the following actions:
 - For a native SQL procedure, define the procedure with the ALLOW DEBUG MODE option and the WLM ENVIRONMENT FOR DEBUG MODE option. If the procedure already exists, you can use the ALTER PROCEDURE statement to specify these options.
 - For an external SQL procedure, use DSNTPSMP or IBM Data Studio to build the SQL procedure with the BUILD_DEBUG option.
 - For a Java stored procedure, define the procedure with the ALLOW DEBUG MODE option, select an appropriate WLM environment for Java debugging, and compile the Java code with the -G option.
 - c. Grant the DEBUGSESSION privilege to the user who runs the debug client.
2. Include breakpoints in your routines or executable files.
3. Follow the instructions for debugging stored procedures in the information for IBM Data Studio.

Related concepts:

 [Java stored procedures and user-defined functions \(DB2 Application Programming for Java\)](#)

Related tasks:

[“Creating an external SQL procedure by using DSNTPSMP”](#) on page 601


 [Developing database routines \(IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio\)](#)

Related reference:

[“Sample programs to help you prepare and run external SQL procedures”](#) on page 614

 [ALTER PROCEDURE \(SQL - native\) \(DB2 SQL\)](#)

 [CREATE PROCEDURE \(SQL - native\) \(DB2 SQL\)](#)

 [The Unified Debugger \(DB2 9 for z/OS Stored Procedures: Through the CALL and Beyond\)](#)

Debugging stored procedures with the Debug Tool for z/OS

You can use the Debug Tool to test z/OS stored procedures that are written in any of the compiled languages that the Debug Tool supports. You can test these stored procedures either interactively or in batch mode.

Using Debug Tool interactively: To test a stored procedure interactively using the Debug Tool, you must have the Debug Tool installed on the z/OS system where the stored procedure runs.

To debug your stored procedure using the Debug Tool:

1. Compile the stored procedure with option TEST. This places information in the program that the Debug Tool uses during a debugging session.
2. Invoke the Debug Tool. One way to do that is to specify the Language Environment run time option TEST. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run time options is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

For example, you can code the TEST option using the following parameters:

```
TEST(ALL,*,PROMPT,JBONES%SESSNA:)
```

The following table lists the effects that each parameter has on the Debug Tool:

Table 166. Effects of the TEST option parameters on the Debug Tool

Parameter value	Effect on the Debug Tool
ALL	The Debug Tool gains control when an attention interrupt, ABEND, or program or Language Environment condition of Severity 1 and above occurs. Debug commands will be entered from the terminal.
PROMPT	The Debug Tool is invoked immediately after Language Environment initialization.
JBONES%SESSNA:	The Debug Tool initiates a session on a workstation identified to APPC/MVS as JBJONES with a session ID of SESSNA.

3. If you want to save the output from your debugging session, issue the following command:

```
SET LOG ON FILE dbgtool.log;
```

This command saves a log of your debugging session to a file on the workstation called dbgtool.log. This should be the first command that you enter from the terminal or include in your commands file.

Using Debug Tool in batch mode: To test your stored procedure in batch mode, you must have the Debug Tool installed on the z/OS system where the stored procedure runs. To debug your stored procedure in batch mode using the Debug Tool, complete the following steps:

- Compile the stored procedure with option TEST, if you plan to use the Language Environment run time option TEST to invoke the Debug Tool. This places information in the program that the Debug Tool uses during a debugging session.
- Allocate a log data set to receive the output from the Debug Tool. Put a DD statement for the log data set in the start-up procedure for the stored procedures address space.
- Enter commands in a data set that you want the Debug Tool to execute. Put a DD statement for that data set in the start-up procedure for the stored procedures address space. To define the commands data set to the Debug Tool, specify the commands data set name or DD name in the TEST run time option.

For example, to specify that the Debug Tool use the commands that are in the data set that is associated with the DD name TESTDD, include the following parameter in the TEST option:

```
TEST(ALL,TESTDD,PROMPT,*)
```

The first command in the commands data set should be:

```
SET LOG ON FILE ddname;
```

This command directs output from your debugging session to the log data set that you defined in the previous step. For example, if you defined a log data set with DD name INSPLOG in the stored procedures address space start-up procedure, the first command should be the following command:

```
SET LOG ON FILE INSPLOG;
```

- Invoke the Debug Tool. The following are two possible methods for invoking the Debug Tool:
 - Specify the run time option TEST. The most convenient place to do that is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.
 - Put CEETEST calls in the stored procedure source code. If you use this approach for an existing stored procedure, you must recompile, re-link, and bind it, and issue the STOP PROCEDURE and START PROCEDURE commands to reload the stored procedure.

You can combine the run time option TEST with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

Related reference:

 [Debug Tool for z/OS](#)

Recording stored procedure debugging messages in a file

You can debug external stored procedures and external SQL procedures by recording debugging messages in a disk file or in a JES spool file. You cannot use this debugging technique for native SQL procedures or Java stored procedures.

To record stored procedure debugging messages in a file:

1. Specify the Language Environment (LE) MSGFILE run time option for the stored procedure. This option identifies where LE is to write the debugging messages. To specify this option, include the RUN OPTIONS clause in either the CREATE PROCEDURE statement or an ALTER PROCEDURE statement.

Specify the following MSGFILE parameters:

- Use the first MSGFILE parameter to specify the JCL DD statement that identifies the data set for the debugging messages. You can direct debugging messages to a disk file or JES spool file. To prevent multiple procedures from sharing a data set, ensure that you specify a unique DD statement.
 - Use the ENQ option to serialize I/O to the message file. This action is necessary, because multiple TCBS can be active in the stored procedure address space. Alternatively, if you debug your applications infrequently or on a DB2 test system, you can serialize I/O by temporarily running the stored procedures address space with NUMTCB=1 in the stored procedures address space start-up procedure.
2. For each instance of MSGFILE that you specify, add a DD statement to the JCL procedure that is used to start the stored procedures address space.

Related reference:

 [ALTER PROCEDURE \(external\) \(DB2 SQL\)](#)

- ALTER PROCEDURE (SQL - external) (DB2 SQL)
- CREATE PROCEDURE (external) (DB2 SQL)
- CREATE PROCEDURE (SQL - external) (DB2 SQL)
- GRANT (system privileges) (DB2 SQL)
- Using Language Environment MSGFILE (z/OS Language Environment Programming Guide)

Driver applications for debugging procedures

You can write a small driver application that calls the stored procedure as a subprogram and passes the parameter list that the stored procedure supports. You can then test and debug the stored procedure as a normal DB2 application under TSO.

Using this method, you can use TSO TEST and other commonly used debugging tools.

Restriction: You cannot use this technique for SQL procedures

DB2 tables that contain debugging information

You can use SQL statements to insert debugging information into a DB2 table. Inserting this information into a table enables other machines in the network (such as a workstation) to easily access the data in the table by using DRDA access.

DB2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.

Debugging an application program

Many sites have guidelines regarding what to do if a program abnormally terminates.

For information about the compiler or assembler test facilities, see the publications for the compiler or CODE/370. The compiler publications include information about the appropriate debugger for the language you are using.

You can also use ISPF Dialog Test to debug your program. You can run all or portions of your application, examine the results, make changes, and rerun it.

Related reference:

- ISPF User's Guide Vol II (z/OS V1R7.0 ISPF User's Guide Vol II)

Locating the problem in an application

If your program does not run correctly, you need to isolate the problem. You should check several items.

Those items are:

- Output from the precompiler, which consists of errors and warnings. Ensure that you have resolved all errors and warnings.

- Output from the compiler or assembler. Ensure that you have resolved all error messages.
- Output from the linkage editor.
 - Have you resolved all external references?
 - Have you included all necessary modules in the correct order?
 - Did you include the correct language interface module? The correct language interface module is:
 - DSNELI or DSNULI for TSO
 - DFSLI000 for IMS
 - DSNCLI or DSNULI for CICS
 - DSNALI or DSNULI for the call attachment facility
 - DSNRLI or DSNULI for the Resource Recovery Services attachment facility
 - Did you specify the correct entry point to your program?
- Output from the bind process.
 - Have you resolved all error messages?
 - Did you specify a plan name? If not, the bind process assumes that you want to process the DBRM for diagnostic purposes, but that you do not want to produce an application plan.
 - Have you specified all the packages that are associated with the programs that make up the application and their partitioned data set (PDS) names in a single application plan?

- Your JCL.

IMS

- If you are using IMS, have you included the DL/I option statement in the correct format?
 - Have you included the region size parameter in the EXEC statement? Does it specify a region size that is large enough for the required storage for the DB2 interface, the TSO, IMS, or CICS system, and your program?
 - Have you included the names of all data sets (DB2 and non-DB2) that the program requires?
- Your program.

You can also use dumps to help localize problems in your program. For example, one of the more common error situations occurs when your program is running and you receive a message that it abended. In this situation, your test procedure might be to capture a TSO dump. To do so, you must allocate a SYSUDUMP or SYSABEND dump data set before calling DB2. When you press the ENTER key (after the error message and READY message), the system requests a dump. You then need to use the FREE command to deallocate the dump data set.

Error and warning messages from the precompiler

In some circumstances, the statements that the DB2 precompiler generates might produce compiler or assembly error messages. You need to know why the messages occur when you compile DB2-produced source statements.

SYSTEMM output from the precompiler

The SYSTEMM output provides a brief summary of the results from the precompiler, all error messages that the precompiler generated, and the statement that is in error, when possible.

The DB2 precompiler provides SYSTERM output when you allocate the DD name SYSTERM. If you use the program preparation panels to prepare and run your program, DB2I allocates SYSTERM according to the TERM option that you specify.

You can use the line number that is provided in each error message in the SYSTERM output to locate the failing source statement.

Figure 82 shows the format of SYSTERM output.

```
DB2 SQL PRECOMPILER      MESSAGES
DSNH104I E      DSNHPARS LINE 32 COL 26  ILLEGAL SYMBOL "X"  VALID SYMBOLS ARE: , FROM1
SELECT VALUE INTO HIPPO X;2

DB2 SQL PRECOMPILER      STATISTICS
SOURCE STATISTICS3
SOURCE LINES READ: 36
NUMBER OF SYMBOLS: 15
SYMBOL TABLE BYTES EXCLUDING ATTRIBUTES: 1848
THERE WERE 1 MESSAGES FOR THIS PROGRAM.4
THERE WERE 0 MESSAGES SUPPRESSED BY THE FLAG OPTION.5
111664 BYTES OF STORAGE WERE USED BY THE PRECOMPILER.6
RETURN CODE IS 87
```

Figure 82. DB2 precompiler SYSTERM output

Notes:

1. Error message.
2. Source SQL statement.
3. Summary statements of source statistics.
4. Summary statement of the number of errors that were detected.
5. Summary statement that indicates the number of errors that were detected but not printed. This situation might occur if you specify a FLAG option other than I.
6. Storage requirement statement that indicates how many bytes of working storage that the DB2 precompiler actually used to process your source statements. That value helps you determine the storage allocation requirements for your program.
7. Return code: 0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.

SYSPRINT output from the precompiler

SYSPRINT output from the DB2 precompiler shows the results of the precompile operation. This output can also include a list of the options that were used, a source code listing, and a host variable cross-reference listing.

When you use the program preparation panels to prepare and run your program, DB2 allocates SYSPRINT according to TERM option that you specify (on line 12 of the PROGRAM PREPARATION: COMPILE, PRELINK, LINK, AND RUN panel). As an alternative, when you use the DSNH command procedure (CLIST), you can specify PRINT(TERM) to obtain SYSPRINT output at your terminal, or you can specify PRINT(*qualifier*) to place the SYSPRINT output into a data set named *authorizationID.qualifier.PCLIST*. Assuming that you do not specify PRINT as LEAVE, NONE, or TERM, DB2 issues a message when the precompiler finishes, telling you where to find your precompiler listings. This helps you locate your diagnostics quickly and easily.

The SYSPRINT output can provide information about your precompiled source module if you specify the options SOURCE and XREF when you start the DB2 precompiler.

The format of SYSPRINT output is as follows:

- A list of the DB2 precompiler options that are in effect during the precompilation (if you did not specify NOOPTIONS).
- A list of your source statements (only if you specified the SOURCE option). An example is shown in Figure 83 on page 1063.
- A list of the symbolic names used in SQL statements (this listing appears only if you specify the XREF option). An example is shown in Figure 84 on page 1063.
- A summary of the errors that are detected by the DB2 precompiler and a list of the error messages that are generated by the precompiler. An example is shown in

The following code shows an example list of DB2 precompiler options as it is displayed in the SYSPRINT output.

```
DB2 SQL PRECOMPILER VERSION 10 REL. 1.0

OPTIONS SPECIFIED: HOST(PLI),SOURCE,XREF,STDSQL(NO),TWOPASS
DSNHDECP LOADED FROM - (USER99.RELM.TESTLIB(DSNHDECP))
OPTIONS USED - SPECIFIED OR DEFAULTED
APOST
APOSTSQL
ATTACH(TSO)
CCSID(37)
CONNECT(2)
DEC(15)
FLAG(I)
FLOAT(S390)
HOST(PLI)
LINECOUNT(60)
MARGINS(2,72)
NEWFUN(V10)
OPTIONS
PERIOD
SOURCE
SQL(DB2)
STDSQL(NO)
TWOPASS
XREF
```

Notes:

1. This section lists the options that are specified at precompilation time. This list does not appear if one of the precompiler option is NOOPTIONS.
2. This section lists the options that are in effect, including defaults, forced values, and options that you specified. The DB2 precompiler overrides or ignores any options that you specify that are inappropriate for the host language.

The following figure shows an example list of source statements as it is displayed in the SYSPRINT output.


```

DB2 SQL PRECOMPILER          TMN5P40:PROCEDURE OPTIONS (MAIN):          PAGE 2

   1      TMN5P40:PROCEDURE OPTIONS(MAIN) ;                               00000100
   2      /*****                                                             00000200
   3      *      program description and prologue                          00000300

:
1324      /*****                                                             00132400
1325      /* GET INFORMATION ABOUT THE PROJECT FROM THE */                 00132500
1326      /* PROJECT TABLE. */                                           00132600
1327      /*****                                                             00132700
1328      EXEC SQL SELECT ACTNO, PREQPROJ, PREQACT                          00132800
1329      INTO PROJ_DATA                                                    00132900
1330      FROM TPREREQ                                                       00133000
1331      WHERE PROJNO = :PROJ_NO;                                          00133100
1332      /*****                                                             00133200
1333      /*****                                                             00133300
1334      /* PROJECT IS FINISHED. DELETE IT. */                             00133400
1335      /*****                                                             00133500
1336      EXEC SQL DELETE FROM PROJ                                         00133600
1337      WHERE PROJNO = :PROJ_NO;                                          00133700
1338

:
1523      END;                                                             00152300

```

Figure 83. DB2 precompiler SYSPRINT output: Source statements section

Notes:

- The left column of sequence numbers, which the DB2 precompiler generates, is for use with the symbol cross-reference listing, the precompiler error messages, and the BIND error messages.
- The right column shows sequence numbers that come from the sequence numbers that are supplied with your source statements.

The following figure shows an example list of symbolic names as it is displayed in the SYSPRINT output.

```

DB2 SQL PRECOMPILER          SYMBOL CROSS-REFERENCE LISTING          PAGE 29

DATA NAMES                    DEFN      REFERENCE

"ACTNO"                       ****     FIELD
                                1328
"PREQACT"                     ****     FIELD
                                1328
"PREQPROJ"                    ****     FIELD
                                1328
"PROJNO"                      ****     FIELD
                                1331 1338

...

PROJ_DATA                     495     CHARACTER(35)
                                1329
PROJ_NO                       496     CHARACTER(3)
                                1331 1338
"TPREREQ"                    ****     TABLE
                                1330 1337

```

Figure 84. DB2 precompiler SYSPRINT output: Symbol cross-reference section

Notes:

DATA NAMES

Identifies the symbolic names that are used in source statements. Names

enclosed in double quotation marks (") or apostrophes (') are names of SQL entities such as tables, columns, and authorization IDs. Other names are host variables.

DEFN

Is the number of the line that the precompiler generates to define the name. **** means that the object was not defined, or the precompiler did not recognize the declarations.

REFERENCE

Contains two kinds of information: the symbolic name, which the source program defines, and which lines refer to the symbolic name. If the symbolic name refers to a valid host variable, the list also identifies the data type or the word STRUCTURE.

The following code shows an example summary report of errors as it is displayed in the SYSPRINT output.

```
DB2 SQL PRECOMPILER          STATISTICS

SOURCE STATISTICS
SOURCE LINES READ: 15231
NUMBER OF SYMBOLS: 1282
SYMBOL TABLE BYTES EXCLUDING ATTRIBUTES: 64323

THERE WERE 1 MESSAGES FOR THIS PROGRAM.4
THERE WERE 0 MESSAGES SUPPRESSED.5
65536 BYTES OF STORAGE WERE USED BY THE PRECOMPILER.6
RETURN CODE IS 8.7
DSNH104I E LINE 590 COL 64 ILLEGAL SYMBOL: 'X'; VALID SYMBOLS ARE: ,FROM8
```

Notes:

1. Summary statement that indicates the number of source lines.
2. Summary statement that indicates the number of symbolic names in the symbol table (SQL names and host names).
3. Storage requirement statement that indicates the number of bytes for the symbol table.
4. Summary statement that indicates the number of messages that are printed.
5. Summary statement that indicates the number of errors that are detected but not printed. You might get this statement if you specify the option FLAG.
6. Storage requirement statement that indicates the number of bytes of working storage that are actually used by the DB2 precompiler to process your source statements.
7. Return code 0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.
8. Error messages (this example detects only one error).

Techniques for debugging programs in TSO

Documenting the errors that are identified during testing of a TSO application helps you investigate and correct problems in the program.

The following information can be useful:

- The application plan name of the program
- The input data that is being processed
- The failing SQL statement and its function

- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The abend code and any error messages

When your program encounters an error that does not result in an abend, it can pass all the required error information to a standard error routine. Online programs might also send an error message to the terminal.

The TSO TEST command

The TSO TEST command is especially useful for debugging assembler programs.

The following example is a command procedure (CLIST) that runs a DB2 application named MYPROG under TSO TEST, and sets an address stop at the entry to the program. The DB2 subsystem name in this example is DB4.

```
PROC 0
TEST 'prefix.SDSNLOAD(DSN)' CP
DSN SYSTEM(DB4)
AT MYPROG.MYPROG.+0 DEFER
GO
RUN PROGRAM(MYPROG) LIBRARY('L186331.RUNLIB.LOAD(MYPROG)')
```

Related reference:

[TEST command \(TSO/E Command Reference\)](#)

Techniques for debugging programs in IMS

Documenting the errors that are identified during testing of an IMS application helps you investigate and correct problems in the program.

The following information can be useful:

- The application plan name for the program
- The input message that is being processed
- The name of the originating logical terminal
- The failing statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The PSB name for the program
- The transaction code that the program was processing
- The call function (that is, the name of a DL/I function)
- The contents of the PCB that the program call refers to
- If a DL/I database call was running, the SSAs, if any, that the call used
- The abend completion code, abend reason code, and any dump error messages

When your program encounters an error, it can pass all the required error information to a standard error routine. Online programs can also send an error message to the originating logical terminal.

An interactive program also can send a message to the master terminal operator giving information about the termination of the program. To do that, the program places the logical terminal name of the master terminal in an express PCB and issues one or more ISRT calls.

Some organizations run a BMP at the end of the day to list all the errors that occurred during the day. If your organization does this, you can send a message by using an express PCB that has its destination set for that BMP.

Batch Terminal Simulator: The Batch Terminal Simulator (BTS) enables you to test IMS application programs. BTS traces application program DL/I calls and SQL statements, and it simulates data communication functions. It can make a TSO terminal appear as an IMS terminal to the terminal operator, which enables the user to interact with the application as though it were an online application. The user can use any application program that is under the user's control to access any database (whether DL/I or DB2) that is under the user's control. Access to DB2 databases requires BTS to operate in batch BMP or TSO BMP mode.

Techniques for debugging programs in CICS

Documenting the errors that are identified during testing of a CICS application helps you investigate and correct problems in the program.

The following information can be useful:

- The application plan name of the program
- The input data that is being processed
- The ID of the originating logical terminal
- The failing SQL statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- Data that is peculiar to CICS that you should record
- Abend code and dump error messages
- Transaction dump, if produced

Using CICS facilities, you can have a printed error record; you can also print the SQLCA and SQLDA contents.

Debugging aids for CICS

CICS provides the following aids to the testing, monitoring, and debugging of application programs:

- **Execution (Command Level) Diagnostic Facility (EDF).** EDF shows CICS commands for all releases of CICS.
- **Abend recovery.** You can use the HANDLE ABEND command to deal with abend conditions. You can use the ABEND command to cause a task to abend.
- **Trace facility.** A trace table can contain entries showing the execution of various CICS commands, SQL statements, and entries that are generated by application programs; you can have these entries written to main storage and, optionally, to an auxiliary storage device.
- **Dump facility.** You can specify areas of main storage to dump onto a sequential data set, either tape or disk, for subsequent offline formatting and printing with a CICS utility program.
- **Journals.** For statistical or monitoring purposes, facilities can create entries in special data sets called journals. The system log is a journal.
- **Recovery.** When an abend occurs, CICS restores certain resources to their original state so that the operator can easily resubmit a transaction for restart.

You can use the SYNCPOINT command to subdivide a program so that you only need to resubmit the uncompleted part of a transaction.

CICS execution diagnostic facility

The CICS execution diagnostic facility (EDF) traces SQL statements in an interactive debugging mode, enabling application programmers to test and debug programs online without changing the program or the program preparation procedure.

EDF intercepts the running application program at various points and displays helpful information about the statement type, input and output variables, and any error conditions after the statement executes. It also displays any screens that the application program sends, so that you can converse with the application program during testing just as a user would on a production system.

EDF displays essential information before and after an SQL statement runs, while the task is in EDF mode. This can be a significant aid in debugging CICS transaction programs that contains SQL statements. The SQL information that EDF displays is helpful for debugging programs and for error analysis after an SQL error or warning. Using this facility reduces the amount of work that you need to do to write special error handlers.

EDF before execution

The following figure shows an example of an EDF screen before it executes an SQL statement. The names of the key information fields on this panel are in **boldface**.

```

TRANSACTION: XC05  PROGRAM: TESTC05  TASK NUMBER: 0000668  DISPLAY: 00
STATUS: ABOUT TO EXECUTE COMMAND
CALL TO RESOURCE MANAGER DSNCSQL
EXEC SQL INSERT
DBRM=TESTC05, STMT=00368, SECT=00004
IVAR 001: TYPE=CHAR,          LEN=00007,   IND=000   AT X'03C92810'
          DATA=X'F0F0F9F4F3F4F2'
IVAR 002: TYPE=CHAR,          LEN=00007,   IND=000   AT X'03C92817'
          DATA=X'F0F1F3F3F7F5F1'
IVAR 003: TYPE=CHAR,          LEN=00004,   IND=000   AT X'03C9281E'
          DATA=X'E7C3F0F5'
IVAR 004: TYPE=CHAR,          LEN=00040,   IND=000   AT X'03C92822'
          DATA=X'E3C5E2E3C3F0F540E2C9D4D7D3C540C4C2F240C9D5E2C5D9E3404040'...
IVAR 005: TYPE=SMALLINT,      LEN=00002,   IND=000   AT X'03C9284A'
          DATA=X'0001'

OFFSET:X'001ECE'  LINE:UNKNOWN  EIBFN=X'1002'

ENTER: CONTINUE
PF1 : UNDEFINED      PF2 : UNDEFINED      PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE  PF6 : USER DISPLAY
PF7 : SCROLL BACK     PF8 : SCROLL FORWARD  PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY  PF11: UNDEFINED       PF12: ABEND USER TASK

```

Figure 85. EDF screen before a DB2 SQL statement

The DB2 SQL information in this screen is as follows:

- EXEC SQL *statement type*
This is the type of SQL statement to execute. The SQL statement can be any valid SQL statement.
- DBRM=*dbrm name*

The name of the database request module (DBRM) that is currently processing. The DBRM, created by the DB2 precompiler, contains information about an SQL statement.

- *STMT=statement number*

This is the DB2 precompiler-generated statement number. The source and error message listings from the precompiler use this statement number, and you can use the statement number to determine which statement is processing. This number is a source line counter that includes host language statements. A statement number that is greater than 32 767 displays as 0.

- *SECT=section number*

The section number of the plan that the SQL statement uses.

SQL statements that contain input host variables

The IVAR (input host variables) section and its attendant fields appear only when the executing statement contains input host variables.

The host variables section includes the variables from predicates, the values used for inserting or updating, and the text of dynamic SQL statements that are being prepared. The address of the input variable is AT 'X'nnnnnnnn'.

Additional host variable information:

- *TYPE=data type*

Specifies the data type for this host variable. The basic data types include character string, graphic string, binary integer, floating-point, decimal, date, time, and timestamp.

- *LEN=length*

Specifies the length of the host variable.

- *IND=indicator variable status number*

Specifies the indicator variable that is associated with this particular host variable. A value of zero indicates that no indicator variable exists. If the value for the selected column is null, DB2 puts a negative value in the indicator variable for this host variable.

- *DATA=host variable data*

Specifies the data, displayed in hexadecimal format, that is associated with this host variable. If the data exceeds what can display on a single line, three periods (...) appear at the far right to indicate that more data is present.

EDF after execution

The following figure shows an example of the first EDF screen that is displayed after the executing an SQL statement. The names of the key information fields on this panel are in **boldface**.

```

TRANSACTION: XC05 PROGRAM: TESTC05 TASK NUMBER: 0000698 DISPLAY: 00
STATUS: COMMAND EXECUTION COMPLETE
CALL TO RESOURCE MANAGER DSNCSQL
EXEC SQL FETCH P.AUTH=SYSADM , S.AUTH=
PLAN=TESTC05, DBRM=TESTC05, STMT=00346, SECT=00001
SQL COMMUNICATION AREA:
SQLCABC = 136 AT X'03C92789'
SQLCODE = 000 AT X'03C9278D'
SQLERRML = 000 AT X'03C92791'
SQLERRMC = '' AT X'03C92793'
SQLERRP = 'DSN' AT X'03C927D9'
SQLERRD(1-6) = 000, 000, 00000, -1, 00000, 000 AT X'03C927E1'
SQLWARN(0-A) = '-----' AT X'03C927F9'
SQLSTATE = 00000 AT X'03C92804'
+ OVAR 001: TYPE=INTEGER, LEN=00004, IND=000 AT X'03C928A0'
DATA=X'00000001'
OFFSET:X'001D14' LINE:UNKNOWN EIBFN=X'1802'

ENTER: CONTINUE
PF1 : UNDEFINED PF2 : UNDEFINED PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS PF5 : WORKING STORAGE PF6 : USER DISPLAY
PF7 : SCROLL BACK PF8 : SCROLL FORWARD PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY PF11: UNDEFINED PF12: ABEND USER TASK

```

Figure 86. EDF screen after a DB2 SQL statement

The DB2 SQL information in this screen is as follows:

- P.AUTH=*primary authorization ID*
The primary DB2 authorization ID.
- S.AUTH=*secondary authorization ID*
The secondary authorization ID. If the RACF list of group options is not active, DB2 uses the connected group name that the CICS attachment facility supplies as the secondary authorization ID. If the RACF list of group options is active, DB2 ignores the connected group name that the CICS attachment facility supplies, but the value is displayed in the DB2 list of secondary authorization IDs.
- PLAN=*plan name*
The name of the plan that is currently running. The PLAN represents the control structure that is produced during the bind process and that is used by DB2 to process SQL statements that are encountered while the application is running.
- SQL Communication Area (SQLCA)
Information in the SQLCA. The SQLCA contains information about errors, if any occur. DB2 uses the SQLCA to give an application program information about the executing SQL statements.

Plus signs (+) on the left of the screen indicate that you can see additional EDF output by using PF keys to scroll the screen forward or back.

The OVAR (output host variables) section and its attendant fields are displayed only when the executing statement returns output host variables.

The following figure contains the rest of the EDF output for this example.

Chapter 20. DB2 sample applications and data

DB2 provides sample data and applications that you can use to learn about DB2 capabilities. DB2 also provides models for your own situations.

Related reference:

 [DB2 sample tables \(Introduction to DB2 for z/OS\)](#)

DB2 sample tables

Much of the DB2 information refers to or relies on the DB2 sample tables. As a group, the tables include information that describes employees, departments, projects, and activities, and they make up a sample application that exemplifies many of the features of DB2.

GUPI

The sample storage group, databases, table spaces, tables, and views are created when you run the installation sample jobs DSNTEJ1 and DSNTEJ7. DB2 sample objects that include LOBs are created in job DSNTEJ7. All other sample objects are created in job DSNTEJ1. The CREATE INDEX statements for the sample tables are not shown here; they, too, are created by the DSNTEJ1 and DSNTEJ7 sample jobs.

Authorization on all sample objects is given to PUBLIC in order to make the sample programs easier to run. You can review the contents of any table by executing an SQL statement, for example `SELECT * FROM DSN8A10.PROJ`. For convenience in interpreting the examples, the department and employee tables are listed in full.

GUPI

Related concepts:

 [Phase 1: Creating and loading sample tables \(DB2 Installation and Migration\)](#)

Activity table (DSN8A10.ACT)

The activity table describes the activities that can be performed during a project.

GUPI

The activity table resides in database DSN8D10A and is created with the following statement:

```
CREATE TABLE DSN8A10.ACT
  (ACTNO    SMALLINT      NOT NULL,
   ACTKWD   CHAR(6)       NOT NULL,
   ACTDESC  VARCHAR(20)   NOT NULL,
   PRIMARY KEY (ACTNO)
)
IN DSN8D10A.DSN8S10P
CCSID EBCDIC;
```

GUPI

Content of the activity table

The following table shows the content of the columns in the activity table.

Table 167. Columns of the activity table

Column	Column name	Description
1	ACTNO	Activity ID (the primary key)
2	ACTKWD	Activity keyword (up to six characters)
3	ACTDESC	Activity description

The activity table has the following indexes.

Table 168. Indexes of the activity table

Name	On column	Type of index
DSN8A10.XACT1	ACTNO	Primary, ascending
DSN8A10.XACT2	ACTKWD	Unique, ascending

Relationship to other tables

The activity table is a parent table of the project activity table, through a foreign key on column ACTNO.

Department table (DSN8A10.DEPT)

The department table describes each department in the enterprise and identifies its manager and the department to which it reports.

GUPI

The department table resides in table space DSN8D10A.DSN8S10D and is created with the following statement:

```
CREATE TABLE DSN8A10.DEPT
  (DEPTNO CHAR(3) NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   MGRNO CHAR(6) ,
   ADMRDEPT CHAR(3) NOT NULL,
   LOCATION CHAR(16) ,
   PRIMARY KEY (DEPTNO) )
IN DSN8D10A.DSN8S10D
CCSID EBCDIC;
```

Because the department table is self-referencing, and also is part of a cycle of dependencies, its foreign keys must be added later with the following statements:

```
ALTER TABLE DSN8A10.DEPT
  FOREIGN KEY RDD (ADMRDEPT) REFERENCES DSN8A10.DEPT
  ON DELETE CASCADE;
```

```
ALTER TABLE DSN8A10.DEPT
  FOREIGN KEY RDE (MGRNO) REFERENCES DSN8A10.EMP
  ON DELETE SET NULL;
```

GUPI

Content of the department table

The following table shows the content of the columns in the department table.

Table 169. Columns of the department table

Column	Column name	Description
1	DEPTNO	Department ID, the primary key.
2	DEPTNAME	A name that describes the general activities of the department.
3	MGRNO	Employee number (EMPNO) of the department manager.
4	ADMRDEPT	ID of the department to which this department reports; the department at the highest level reports to itself.
5	LOCATION	The remote location name.

The following table shows the indexes of the department table.

Table 170. Indexes of the department table

Name	On column	Type of index
DSN8A10.XDEPT1	DEPTNO	Primary, ascending
DSN8A10.XDEPT2	MGRNO	Ascending
DSN8A10.XDEPT3	ADMRDEPT	Ascending

The following table shows the content of the department table.

Table 171. DSN8A10.DEPT: department table

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	-----
B01	PLANNING	000020	A00	-----
C01	INFORMATION CENTER	000030	A00	-----
D01	DEVELOPMENT CENTER	-----	A00	-----
E01	SUPPORT SERVICES	000050	A00	-----
D11	MANUFACTURING SYSTEMS	000060	D01	-----
D21	ADMINISTRATION SYSTEMS	000070	D01	-----
E11	OPERATIONS	000090	E01	-----
E21	SOFTWARE SUPPORT	000100	E01	-----
F22	BRANCH OFFICE F2	-----	E01	-----
G22	BRANCH OFFICE G2	-----	E01	-----
H22	BRANCH OFFICE H2	-----	E01	-----
I22	BRANCH OFFICE I2	-----	E01	-----
J22	BRANCH OFFICE J2	-----	E01	-----

The LOCATION column contains null values until sample job DSNTEJ6 updates this column with the location name.

Relationship to other tables

The department table is self-referencing: the value of the administering department must be a valid department ID.

The department table is a parent table of the following :

- The employee table, through a foreign key on column WORKDEPT
- The project table, through a foreign key on column DEPTNO

The department table is a dependent of the employee table, through its foreign key on column MGRNO.

Employee table (DSN8A10.EMP)

The sample employee table identifies all employees by an employee number and lists basic personnel information.

GUIP The employee table resides in the partitioned table space DSN8D10A.DSN8S10E. Because this table has a foreign key that references DEPT, that table and the index on its primary key must be created first. Then EMP is created with the following statement:

```
CREATE TABLE DSN8A10.EMP
  (EMPNO      CHAR(6)                NOT NULL,
   FIRSTNME   VARCHAR(12)           NOT NULL,
   MIDINIT    CHAR(1)                NOT NULL,
   LASTNAME   VARCHAR(15)           NOT NULL,
   WORKDEPT   CHAR(3)                ,
   PHONENO    CHAR(4)                CONSTRAINT NUMBER CHECK
   (PHONENO >= '0000' AND
    PHONENO <= '9999')              ,
   HIREDATE   DATE                   ,
   JOB        CHAR(8)                 ,
   EDLEVEL    SMALLINT                ,
   SEX        CHAR(1)                 ,
   BIRTHDATE  DATE                   ,
   SALARY     DECIMAL(9,2)            ,
   BONUS      DECIMAL(9,2)            ,
   COMM       DECIMAL(9,2)            ,
   PRIMARY KEY (EMPNO)                ,
   FOREIGN KEY REF (WORKDEPT) REFERENCES DSN8A10.DEPT
   ON DELETE SET NULL                 )
EDITPROC DSN8EAE1
IN DSN8D10A.DSN8S10E
CCSID EBCDIC;
```

GUIP

Content of the employee table

The following table shows the type of content of each of the columns in the employee table. The table has a check constraint, NUMBER, which checks that the four-digit phone number is in the numeric range 0000 to 9999.

Table 172. Columns of the employee table

Column	Column name	Description
1	EMPNO	Employee number (the primary key)
2	FIRSTNME	First name of employee
3	MIDINIT	Middle initial of employee

Table 172. Columns of the employee table (continued)

Column	Column name	Description
4	LASTNAME	Last name of employee
5	WORKDEPT	ID of department in which the employee works
6	PHONENO	Employee telephone number
7	HIREDATE	Date of hire
8	JOB	Job held by the employee
9	EDLEVEL	Number of years of formal education
10	SEX	Sex of the employee (M or F)
11	BIRTHDATE	Date of birth
12	SALARY	Yearly salary in dollars
13	BONUS	Yearly bonus in dollars
14	COMM	Yearly commission in dollars

The following table shows the indexes of the employee table.

Table 173. Indexes of the employee table

Name	On column	Type of index
DSN8A10.XEMP1	EMPNO	Primary, partitioned, ascending
DSN8A10.XEMP2	WORKDEPT	Ascending

The following table shows the first half (left side) of the content of the employee table. (Table 175 on page 1076 shows the remaining content (right side) of the employee table.)

Table 174. Left half of DSN8A10.EMP: employee table. Note that a blank in the MIDINIT column is an actual value of " " rather than null.

EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10
000030	SALLY	A	KWAN	C01	4738	1975-04-05
000050	JOHN	B	GEYER	E01	6789	1949-08-17
000060	IRVING	F	STERN	D11	6423	1973-09-14
000070	EVA	D	PULASKI	D21	7831	1980-09-30
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19
000110	VINCENZO	G	LUCCHESI	A00	3490	1958-05-16
000120	SEAN		O'CONNELL	A00	2167	1963-12-05
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15
000150	BRUCE		ADAMSON	D11	4510	1972-02-12
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07
000190	JAMES	H	WALKER	D11	2986	1974-07-26
000200	DAVID		BROWN	D11	4501	1966-03-03
000210	WILLIAM	T	JONES	D11	0942	1979-04-11
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29

Table 174. Left half of DSN8A10.EMP: employee table (continued). Note that a blank in the MIDINIT column is an actual value of " " rather than null.

EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1969-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24
000290	JOHN	R	PARKER	E11	4502	1980-05-30
000300	PHILIP	X	SMITH	E11	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5698	1947-05-05
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01
200120	GREG		ORLANDO	A00	2167	1972-05-05
200140	KIM	N	NATZ	C01	1793	1976-12-15
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15
200220	REBA	K	JOHN	D11	0672	1968-08-29
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12
200330	HELENA		WONG	E21	2103	1976-02-23
200340	ROY	R	ALONZO	E21	5698	1947-05-05

(Table 174 on page 1075 shows the first half (right side) of the content of employee table.)

Table 175. Right half of DSN8A10.EMP: employee table

(EMPNO)	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
(000010)	PRES	18	F	1933-08-14	52750.00	1000.00	4220.00
(000020)	MANAGER	18	M	1948-02-02	41250.00	800.00	3300.00
(000030)	MANAGER	20	F	1941-05-11	38250.00	800.00	3060.00
(000050)	MANAGER	16	M	1925-09-15	40175.00	800.00	3214.00
(000060)	MANAGER	16	M	1945-07-07	32250.00	600.00	2580.00
(000070)	MANAGER	16	F	1953-05-26	36170.00	700.00	2893.00
(000090)	MANAGER	16	F	1941-05-15	29750.00	600.00	2380.00
(000100)	MANAGER	14	M	1956-12-18	26150.00	500.00	2092.00
(000110)	SALESREP	19	M	1929-11-05	46500.00	900.00	3720.00
(000120)	CLERK	14	M	1942-10-18	29250.00	600.00	2340.00
(000130)	ANALYST	16	F	1925-09-15	23800.00	500.00	1904.00
(000140)	ANALYST	18	F	1946-01-19	28420.00	600.00	2274.00
(000150)	DESIGNER	16	M	1947-05-17	25280.00	500.00	2022.00
(000160)	DESIGNER	17	F	1955-04-12	22250.00	400.00	1780.00
(000170)	DESIGNER	16	M	1951-01-05	24680.00	500.00	1974.00
(000180)	DESIGNER	17	F	1949-02-21	21340.00	500.00	1707.00
(000190)	DESIGNER	16	M	1952-06-25	20450.00	400.00	1636.00
(000200)	DESIGNER	16	M	1941-05-29	27740.00	600.00	2217.00
(000210)	DESIGNER	17	M	1953-02-23	18270.00	400.00	1462.00
(000220)	DESIGNER	18	F	1948-03-19	29840.00	600.00	2387.00
(000230)	CLERK	14	M	1935-05-30	22180.00	400.00	1774.00
(000240)	CLERK	17	M	1954-03-31	28760.00	600.00	2301.00

Table 175. Right half of DSN8A10.EMP: employee table (continued)

(EMPNO)	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
(000250)	CLERK	15	M	1939-11-12	19180.00	400.00	1534.00
(000260)	CLERK	16	F	1936-10-05	17250.00	300.00	1380.00
(000270)	CLERK	15	F	1953-05-26	27380.00	500.00	2190.00
(000280)	OPERATOR	17	F	1936-03-28	26250.00	500.00	2100.00
(000290)	OPERATOR	12	M	1946-07-09	15340.00	300.00	1227.00
(000300)	OPERATOR	14	M	1936-10-27	17750.00	400.00	1420.00
(000310)	OPERATOR	12	F	1931-04-21	15900.00	300.00	1272.00
(000320)	FIELDREP	16	M	1932-08-11	19950.00	400.00	1596.00
(000330)	FIELDREP	14	M	1941-07-18	25370.00	500.00	2030.00
(000340)	FIELDREP	16	M	1926-05-17	23840.00	500.00	1907.00
(200010)	SALESREP	18	F	1933-08-14	46500.00	1000.00	4220.00
(200120)	CLERK	14	M	1942-10-18	29250.00	600.00	2340.00
(200140)	ANALYST	18	F	1946-01-19	28420.00	600.00	2274.00
(200170)	DESIGNER	16	M	1951-01-05	24680.00	500.00	1974.00
(200220)	DESIGNER	18	F	1948-03-19	29840.00	600.00	2387.00
(200240)	CLERK	17	M	1954-03-31	28760.00	600.00	2301.00
(200280)	OPERATOR	17	F	1936-03-28	26250.00	500.00	2100.00
(200310)	OPERATOR	12	F	1931-04-21	15900.00	300.00	1272.00
(200330)	FIELDREP	14	F	1941-07-18	25370.00	500.00	2030.00
(200340)	FIELDREP	16	M	1926-05-17	23840.00	500.00	1907.00

Relationship to other tables

The employee table is a parent table of:

- The department table, through a foreign key on column MGRNO
- The project table, through a foreign key on column RESPEMP

The employee table is a dependent of the department table, through its foreign key on column WORKDEPT.

Employee photo and resume table (DSN8A10.EMP_PHOTO_RESUME)

The sample employee photo and resume table complements the employee table.

GUIP Each row of the photo and resume table contains a photo of the employee, in two formats, and the employee's resume. The photo and resume table resides in table space DSN8D10A.DSN8S10E. The following statement creates the table:

```
CREATE TABLE DSN8A10.EMP_PHOTO_RESUME
  (EMPNO CHAR(06) NOT NULL,
   EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,
   PSEG_PHOTO BLOB(500K),
   BMP_PHOTO BLOB(100K),
   RESUME CLOB(5K))
  PRIMARY KEY (EMPNO)
  IN DSN8D10L.DSN8S10B
  CCSID EBCDIC;
```

DB2 requires an auxiliary table for each LOB column in a table. The following statements define the auxiliary tables for the three LOB columns in DSN8A10.EMP_PHOTO_RESUME:

```

CREATE AUX TABLE DSN8A10.AUX_BMP_PHOTO
    IN DSN8D10L.DSN8S10M
    STORES DSN8A10.EMP_PHOTO_RESUME
    COLUMN BMP_PHOTO;

CREATE AUX TABLE DSN8A10.AUX_PSEG_PHOTO
    IN DSN8D10L.DSN8S10L
    STORES DSN8A10.EMP_PHOTO_RESUME
    COLUMN PSEG_PHOTO;

CREATE AUX TABLE DSN8A10.AUX_EMP_RESUME
    IN DSN8D10L.DSN8S10N
    STORES DSN8A10.EMP_PHOTO_RESUME
    COLUMN RESUME;

```



Content of the employee photo and resume table

The following table shows the content of the columns in the employee photo and resume table.

Table 176. Columns of the employee photo and resume table

Column	Column name	Description
1	EMPNO	Employee ID (the primary key).
2	EMP_ROWID	Row ID to uniquely identify each row of the table. DB2 supplies the values of this column.
3	PSEG_PHOTO	Employee photo, in PSEG format.
4	BMP_PHOTO	Employee photo, in BMP format.
5	RESUME	Employee resume.

The following table shows the indexes for the employee photo and resume table.

Table 177. Indexes of the employee photo and resume table

Name	On column	Type of index
DSN8A10.XEMP_PHOTO_RESUME	EMPNO	Primary, ascending

The following table shows the indexes for the auxiliary tables that support the employee photo and resume table.

Table 178. Indexes of the auxiliary tables for the employee photo and resume table

Name	On table	Type of index
DSN8A10.XAUX_BMP_PHOTO	DSN8A10.AUX_BMP_PHOTO	Unique
DSN8A10.XAUX_PSEG_PHOTO	DSN8A10.AUX_PSEG_PHOTO	Unique
DSN8A10.XAUX_EMP_RESUME	DSN8A10.AUX_EMP_RESUME	Unique

Relationship to other tables

The employee photo and resume table is a parent table of the project table, through a foreign key on column RESPEMP.

Project table (DSN8A10.PROJ)

The sample project table describes each project that the business is currently undertaking. Data that is contained in each row of the table includes the project number, name, person responsible, and schedule dates.

The project table resides in database DSN8D10A. Because this table has foreign keys that reference DEPT and EMP, those tables and the indexes on their primary keys must be created first. Then PROJ is created with the following statement:

GUPI

```
CREATE TABLE DSN8A10.PROJ
      (PROJNO CHAR(6) PRIMARY KEY NOT NULL,
       PROJNAME VARCHAR(24) NOT NULL WITH DEFAULT
         'PROJECT NAME UNDEFINED',
       DEPTNO CHAR(3) NOT NULL REFERENCES
         DSN8A10.DEPT ON DELETE RESTRICT,
       RESPEMP CHAR(6) NOT NULL REFERENCES
         DSN8A10.EMP ON DELETE RESTRICT,
       PRSTAFF DECIMAL(5, 2) ,
       PRSTDATE DATE ,
       PRENDATE DATE ,
       MAJPROJ CHAR(6))
      IN DSN8D10A.DSN8S10P
      CCSID EBCDIC;
```

Because the project table is self-referencing, the foreign key for that constraint must be added later with the following statement:

```
ALTER TABLE DSN8A10.PROJ
      FOREIGN KEY RPP (MAJPROJ) REFERENCES DSN8A10.PROJ
      ON DELETE CASCADE;
```

GUPI

Content of the project table

The following table shows the content of the columns of the project table.

Table 179. Columns of the project table

Column	Column name	Description
1	PROJNO	Project ID (the primary key)
2	PROJNAME	Project name
3	DEPTNO	ID of department responsible for the project
4	RESPEMP	ID of employee responsible for the project
5	PRSTAFF	Estimated mean number of persons that are needed between PRSTDATE and PRENDATE to complete the whole project, including any subprojects
6	PRSTDATE	Estimated project start date
7	PRENDATE	Estimated project end date
8	MAJPROJ	ID of any project of which this project is a part

The following table shows the indexes for the project table:

Table 180. Indexes of the project table

Name	On column	Type of index
DSN8A10.XPROJ1	PROJNO	Primary, ascending
DSN8A10.XPROJ2	RESPEMP	Ascending

Relationship to other tables

The table is self-referencing; a non-null value of MAJPROJ must be a valid project number. The table is a parent table of the project activity table, through a foreign key on column PROJNO. It is a dependent of the following tables:

- The department table, through its foreign key on DEPTNO
- The employee table, through its foreign key on RESPEMP

Project activity table (DSN8A10.PROJACT)

The sample project activity table lists the activities that are performed for each project.

The project activity table resides in database DSN8D10A. Because this table has foreign keys that reference PROJ and ACT, those tables and the indexes on their primary keys must be created first. Then PROJACT is created with the following statement:

GUIP

```

CREATE TABLE DSN8A10.PROJACT
  (PROJNO  CHAR(6)                NOT NULL,
   ACTNO   SMALLINT              NOT NULL,
   ACSTAFF DECIMAL(5,2)          ,
   ACSTDATE DATE                 NOT NULL,
   ACENDATE DATE                 ,
   PRIMARY KEY (PROJNO, ACTNO, ACSTDATE),
   FOREIGN KEY RPAP (PROJNO) REFERENCES DSN8A10.PROJ
                                     ON DELETE RESTRICT,
   FOREIGN KEY RPAA (ACTNO) REFERENCES DSN8A10.ACT
                                     ON DELETE RESTRICT)

IN DSN8D10A.DSN8S10P
CCSID EBCDIC;
```

GUIP

Content of the project activity table

The following table shows the content of the columns of the project activity table.

Table 181. Columns of the project activity table

Column	Column name	Description
1	PROJNO	Project ID
2	ACTNO	Activity ID
3	ACSTAFF	Estimated mean number of employees that are needed to staff the activity
4	ACSTDATE	Estimated activity start date
5	ACENDATE	Estimated activity completion date

The following table shows the index of the project activity table:

Table 182. Index of the project activity table

Name	On columns	Type of index
DSN8A10.XPROJAC1	PROJNO, ACTNO, ACSTDATE	primary, ascending

Relationship to other tables

The project activity table is a parent table of the employee to project activity table, through a foreign key on columns PROJNO, ACTNO, and EMSTDATE. It is a dependent of the following tables:

- The activity table, through its foreign key on column ACTNO
- The project table, through its foreign key on column PROJNO

Related reference:

“Activity table (DSN8A10.ACT)” on page 1071

“Project table (DSN8A10.PROJ)” on page 1079

Employee-to-project activity table (DSN8A10.EMPPROJACT)

The sample employee-to-project activity table identifies the employee who performs an activity for a project, tells the proportion of the employee's time that is required, and gives a schedule for the activity.

GUIP

The employee-to-project activity table resides in database DSN8D10A. Because this table has foreign keys that reference EMP and PROJACT, those tables and the indexes on their primary keys must be created first. Then EMPPROJACT is created with the following statement:

```
CREATE TABLE DSN8A10.EMPPROJACT
  (EMPNO      CHAR(6)                NOT NULL,
   PROJNO     CHAR(6)                NOT NULL,
   ACTNO      SMALLINT               NOT NULL,
   EMPTIME    DECIMAL(5,2)           ,
   EMSTDATE   DATE                   ,
   EMENDATE   DATE                   ,
   FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
     REFERENCES DSN8A10.PROJACT
     ON DELETE RESTRICT,
   FOREIGN KEY REPAE (EMPNO) REFERENCES DSN8A10.EMP
     ON DELETE RESTRICT)
IN DSN8D10A.DSN8S10P
CCSID EBCDIC;
```

GUIP

Content of the employee-to-project activity table

The following table shows the content of the columns in the employee-to-project activity table.

Table 183. Columns of the employee-to-project activity table

Column	Column name	Description
1	EMPNO	Employee ID number

Table 183. Columns of the employee-to-project activity table (continued)

Column	Column name	Description
2	PROJNO	Project ID of the project
3	ACTNO	ID of the activity within the project
4	EMPTIME	A proportion of the employee's full time (between 0.00 and 1.00) that is to be spent on the activity
5	EMSTDATE	Date the activity starts
6	EMENDATE	Date the activity ends

The following table shows the indexes for the employee-to-project activity table:

Table 184. Indexes of the employee-to-project activity table

Name	On columns	Type of index
DSN8A10.XEMPPROJECT1	PROJNO, ACTNO, EMSTDATE, EMPNO	Unique, ascending
DSN8A10.XEMPPROJECT2	EMPNO	Ascending

Relationship to other tables

The employee-to-project activity table is a dependent of the following tables:

- The employee table, through its foreign key on column EMPNO
- The project activity table, through its foreign key on columns PROJNO, ACTNO, and EMSTDATE.

Related reference:

“Employee table (DSN8A10.EMP)” on page 1074

“Project activity table (DSN8A10.PROJECT)” on page 1080

Unicode sample table (DSN8A10.DEMO_UNICODE)

The Unicode sample table is used to verify that data conversions to and from EBCDIC and Unicode are working as expected.

GUIP

The table resides in database DSN8D10A, and is defined with the following statement:

```
CREATE TABLE DSN8A10.DEMO_UNICODE
  (LOWER_A_TO_Z      CHAR(26)          ,
   UPPER_A_TO_Z     CHAR(26)          ,
   ZERO_TO_NINE     CHAR(10)         ,
   X00_TO_XFF      VARCHAR(256)      FOR BIT DATA)
  IN DSN8D81E.DSN8S81U
  CCSID UNICODE;
```

GUIP

Content of the Unicode sample table

The following table shows the content of the columns in the Unicode sample table:

Table 185. Columns of the Unicode sample table

Column	Column Name	Description
1	LOWER_A_TO_Z	Array of characters, 'a' to 'z'
2	UPPER_A_TO_Z	Array of characters, 'A' to 'Z'
3	ZERO_TO_NINE	Array of characters, '0' to '9'
4	X00_TO_XFF	Array of characters, 'x'00' to 'x'FF'

This table has no indexes.

Relationship to other tables

This table has no relationship to other tables.

Relationships among the sample tables

Relationships among the sample tables are established by foreign keys in dependent tables that reference primary keys in parent tables.

The following figure shows relationships among the sample tables. You can find descriptions of the columns with the descriptions of the tables.

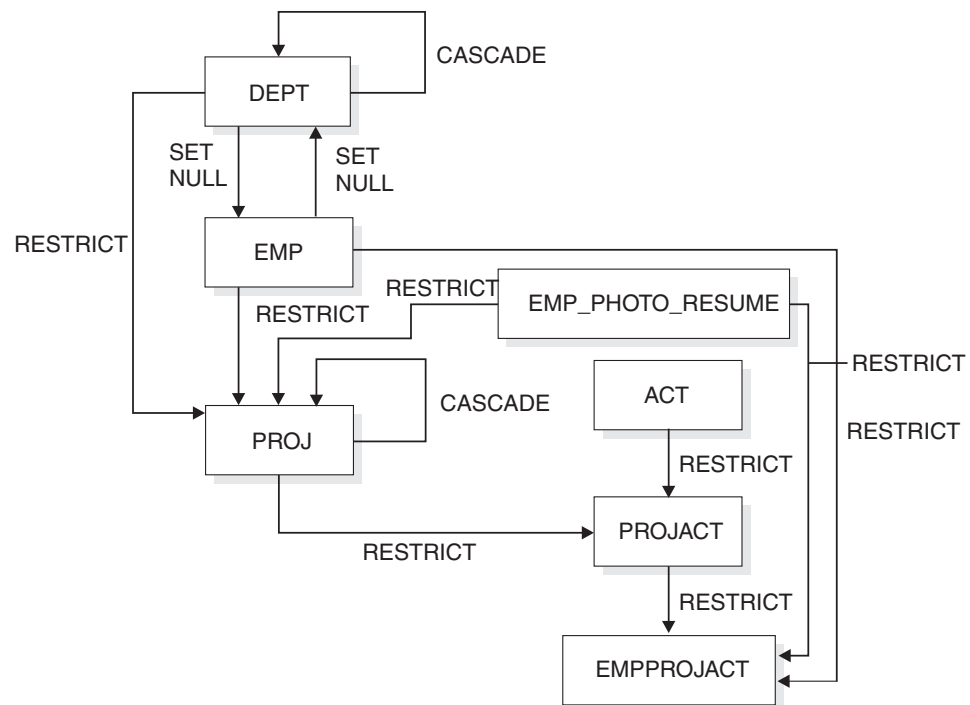


Figure 88. Relationships among tables in the sample application

Related reference:

“Activity table (DSN8A10.ACT)” on page 1071

“Department table (DSN8A10.DEPT)” on page 1072

“Employee photo and resume table (DSN8A10.EMP_PHOTO_RESUME)” on page 1077

“Employee table (DSN8A10.EMP)” on page 1074

“Employee-to-project activity table (DSN8A10.EMPPROJACT)” on page 1081

“Project activity table (DSN8A10.PROJACT)” on page 1080

“Project table (DSN8A10.PROJ)” on page 1079

“Unicode sample table (DSN8A10.DEMO_UNICODE)” on page 1082

Views on the sample tables

DB2 creates a number of views on the sample tables for use in the sample applications.

The following table indicates the tables on which each view is defined and the sample applications that use the view. All view names have the qualifier DSN8A10.

Table 186. Views on sample tables

View name	On tables or views	Used in application
VDEPT	DEPT	Organization Project
VHDEPT	DEPT	Distributed organization
VEMP	EMP	Distributed organization Organization Project
VPROJ	PROJ	Project
VACT	ACT	Project
VPROJACT	PROJACT	Project
VEMPPROJACT	EMPPROJACT	Project
VDEPMG1	DEPT EMP	Organization
VEMPDPT1	DEPT EMP	Organization
VASTRDE1	DEPT	
VASTRDE2	VDEPMG1 EMP	Organization
VPROJRE1	PROJ EMP	Project
VPSTRDE1	VPROJRE1 VPROJRE2	Project
VPSTRDE2	VPROJRE1	Project
VFORPLA	VPROJRE1 EMPPROJACT	Project

Table 186. Views on sample tables (continued)

View name	On tables or views	Used in application
VSTAFAC1	PROJACT ACT	Project
VSTAFAC2	EMPPROJACT ACT EMP	Project
VPHONE	EMP DEPT	Phone
VEMPLP	EMP	Phone

GUPI

The following SQL statement creates the view named VDEPT.

```
CREATE VIEW DSN8A10.VDEPT
  AS SELECT ALL      DEPTNO  ,
                    DEPTNAME,
                    MGRNO   ,
                    ADMRDEPT
  FROM DSN8A10.DEPT;
```

The following SQL statement creates the view named VHDEPT.

```
CREATE VIEW DSN8A10.VHDEPT
  AS SELECT ALL      DEPTNO  ,
                    DEPTNAME,
                    MGRNO   ,
                    ADMRDEPT,
                    LOCATION
  FROM DSN8A10.DEPT;
```

The following SQL statement creates the view named VEMP.

```
CREATE VIEW DSN8A10.VEMP
  AS SELECT ALL      EMPNO   ,
                    FIRSTNME,
                    MIDINIT  ,
                    LASTNAME,
                    WORKDEPT
  FROM DSN8A10.EMP;
```

The following SQL statement creates the view named VPROJ.

```
CREATE VIEW DSN8A10.VPROJ
  AS SELECT ALL
      PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF,
      PRSTDATE, PRENDATE, MAJPROJ
  FROM DSN8A10.PROJ ;
```

The following SQL statement creates the view named VACT.

```
CREATE VIEW DSN8A10.VACT
  AS SELECT ALL      ACTNO   ,
                    ACTKWD  ,
                    ACTDESC
  FROM DSN8A10.ACT ;
```

The following SQL statement creates the view named VPROJACT.

```
CREATE VIEW DSN8A10.VPROJACT
  AS SELECT ALL
      PROJNO,ACTNO, ACSTAFF, ACSTDATE, ACENDATE
  FROM DSN8A10.PROJACT ;
```

The following SQL statement creates the view named VEMPPROJACT.

```
CREATE VIEW DSN8A10.VEMPPROJACT
  AS SELECT ALL
      EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, EMENDATE
  FROM DSN8A10.EMPPROJACT ;
```

The following SQL statement creates the view named VDEPMG1.

```
CREATE VIEW DSN8A10.VDEPMG1
  (DEPTNO, DEPTNAME, MGRNO, FIRSTNME, MIDINIT,
   LASTNAME, ADMRDEPT)
  AS SELECT ALL
      DEPTNO, DEPTNAME, EMPNO, FIRSTNME, MIDINIT,
      LASTNAME, ADMRDEPT
  FROM DSN8A10.DEPT LEFT OUTER JOIN DSN8A10.EMP
  ON MGRNO = EMPNO ;
```

The following SQL statement creates the view named VEMPDPT1.

```
CREATE VIEW DSN8A10.VEMPDPT1
  (DEPTNO, DEPTNAME, EMPNO, FRSTINIT, MIDINIT,
   LASTNAME, WORKDEPT)
  AS SELECT ALL
      DEPTNO, DEPTNAME, EMPNO, SUBSTR(FIRSTNME, 1, 1), MIDINIT,
      LASTNAME, WORKDEPT
  FROM DSN8A10.DEPT RIGHT OUTER JOIN DSN8A10.EMP
  ON WORKDEPT = DEPTNO ;
```

The following SQL statement creates the view named VASTRDE1.

```
CREATE VIEW DSN8A10.VASTRDE1
  (DEPT1NO,DEPT1NAM,EMP1NO,EMP1FN,EMP1MI,EMP1LN,TYPE2,
   DEPT2NO,DEPT2NAM,EMP2NO,EMP2FN,EMP2MI,EMP2LN)
  AS SELECT ALL
      D1.DEPTNO,D1.DEPTNAME,D1.MGRNO,D1.FIRSTNME,D1.MIDINIT,
      D1.LASTNAME, '1',
      D2.DEPTNO,D2.DEPTNAME,D2.MGRNO,D2.FIRSTNME,D2.MIDINIT,
      D2.LASTNAME
  FROM DSN8A10.VDEPMG1 D1, DSN8A10.VDEPMG1 D2
  WHERE D1.DEPTNO = D2.ADMRDEPT ;
```

The following SQL statement creates the view named VASTRDE2.

```
CREATE VIEW DSN8A10.VASTRDE2
  (DEPT1NO,DEPT1NAM,EMP1NO,EMP1FN,EMP1MI,EMP1LN,TYPE2,
   DEPT2NO,DEPT2NAM,EMP2NO,EMP2FN,EMP2MI,EMP2LN)
  AS SELECT ALL
      D1.DEPTNO,D1.DEPTNAME,D1.MGRNO,D1.FIRSTNME,D1.MIDINIT,
      D1.LASTNAME, '2',
      D1.DEPTNO,D1.DEPTNAME,E2.EMPNO,E2.FIRSTNME,E2.MIDINIT,
      E2.LASTNAME
  FROM DSN8A10.VDEPMG1 D1, DSN8A10.EMP E2
  WHERE D1.DEPTNO = E2.WORKDEPT;
```

The following figure shows the SQL statement that creates the view named VPROJRE1.


```

CREATE VIEW DSN8A10.VPROJRE1
(PROJNO,PROJNAME,PROJDEP,RESPEMP,FIRSTNME,MIDINIT,
LASTNAME,MAJPROJ)
AS SELECT ALL
    PROJNO,PROJNAME,DEPTNO,EMPNO,FIRSTNME,MIDINIT,
    LASTNAME,MAJPROJ
FROM DSN8A10.PROJ, DSN8A10.EMP
WHERE RESPEMP = EMPNO ;

```

Figure 89. VPROJRE1

The following SQL statement creates the view named VPSTRDE1.

```

CREATE VIEW DSN8A10.VPSTRDE1
(PROJ1NO,PROJ1NAME,RESP1NO,RESP1FN,RESP1MI,RESP1LN,
PROJ2NO,PROJ2NAME,RESP2NO,RESP2FN,RESP2MI,RESP2LN)
AS SELECT ALL
    P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
    P1.LASTNAME,
    P2.PROJNO,P2.PROJNAME,P2.RESPEMP,P2.FIRSTNME,P2.MIDINIT,
    P2.LASTNAME
FROM DSN8A10.VPROJRE1 P1,
    DSN8A10.VPROJRE1 P2
WHERE P1.PROJNO = P2.MAJPROJ ;

```

The following SQL statement creates the view named VPSTRDE2.

```

CREATE VIEW DSN8A10.VPSTRDE2
(PROJ1NO,PROJ1NAME,RESP1NO,RESP1FN,RESP1MI,RESP1LN,
PROJ2NO,PROJ2NAME,RESP2NO,RESP2FN,RESP2MI,RESP2LN)
AS SELECT ALL
    P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
    P1.LASTNAME,
    P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
    P1.LASTNAME
FROM DSN8A10.VPROJRE1 P1
WHERE NOT EXISTS
    (SELECT * FROM DSN8A10.VPROJRE1 P2
    WHERE P1.PROJNO = P2.MAJPROJ) ;

```

The following SQL statement creates the view named VFORPLA.

```

CREATE VIEW DSN8A10.VFORPLA
(PROJNO,PROJNAME,RESPEMP,PROJDEP,FRSTINIT,MIDINIT,LASTNAME)
AS SELECT ALL
    F1.PROJNO,PROJNAME,RESPEMP,PROJDEP, SUBSTR(FIRSTNME, 1, 1),
    MIDINIT, LASTNAME
FROM DSN8A10.VPROJRE1 F1 LEFT OUTER JOIN DSN8A10.EMPPROJECT F2
ON F1.PROJNO = F2.PROJNO;

```

The following SQL statement creates the view named VSTAFAC1.

```

CREATE VIEW DSN8A10.VSTAFAC1
(PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
EMPTIME,STDATE,ENDATE, TYPE)
AS SELECT ALL
    PA.PROJNO, PA.ACTNO, AC.ACTDESC,' ',' ',' ',' ',
    PA.ACSTAFF, PA.ACSTDATE,
    PA.ACENDATE,'1'
FROM DSN8A10.PROJECT PA, DSN8A10.ACT AC
WHERE PA.ACTNO = AC.ACTNO ;

```

The following SQL statement creates the view named VSTAFAC2.

```

CREATE VIEW DSN8A10.VSTAFAC2
(PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
EMPTIME,STDATE, ENDATE, TYPE)
AS SELECT ALL

```

```

        EP.PROJNO, EP.ACTNO, AC.ACTDESC, EP.EMPNO,EM.FIRSTNME,
        EM.MIDINIT, EM.LASTNAME, EP.EMPTIME, EP.EMSTDATE,
        EP.EMENDATE, '2'
FROM DSN8A10.EMP PROJACT EP, DSN8A10.ACT AC, DSN8A10.EMP EM
WHERE EP.ACTNO = AC.ACTNO AND EP.EMPNO = EM.EMPNO ;

```

The following SQL statement creates the view named VPHONE.

```

CREATE VIEW DSN8A10.VPHONE
    (LASTNAME,
     FIRSTNAME,
     MIDDLEINITIAL,
     PHONENUMBER,
     EMPLOYEEENUNBER,
     DEPTNUMBER,
     DEPTNAME)
AS SELECT ALL      LASTNAME,
                   FIRSTNME,
                   MIDINIT ,
                   VALUE(PHONENO, '  '),
                   EMPNO,
                   DEPTNO,
                   DEPTNAME
FROM DSN8A10.EMP, DSN8A10.DEPT
WHERE WORKDEPT = DEPTNO;

```

The following SQL statement creates the view named VEMPLP.

```

CREATE VIEW DSN8A10.VEMPLP
    (EMPLOYEEENUNBER,
     PHONENUMBER)
AS SELECT ALL      EMPNO ,
                   PHONENO
FROM DSN8A10.EMP ;

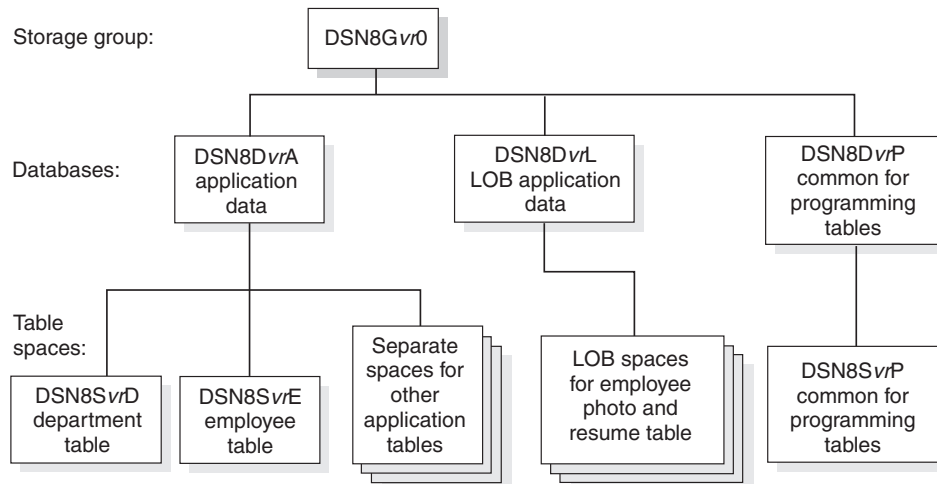
```



Storage of sample application tables

Normally, related data is stored in the same database.

The following figure shows how the sample tables are related to databases and storage groups. Two databases are used to illustrate the possibility.



vr is a 2-digit version identifier.

Figure 90. Relationship among sample databases and table spaces

In addition to the storage group and databases that are shown in the preceding figure, the storage group DSN8G10U and database DSN8D10U are created when you run DSNTJ2A.

Storage group for sample application data

Sample application data is stored in storage group DSN8G100. The default storage group, SYSDEFLT, which is created when DB2 is installed, is not used to store sample application data.

GUIP

The storage group that is used to store sample application data is defined by the following statement:

```
CREATE STOGROUP DSN8G100
  VOLUMES (DSNV01)
  VCAT DSN100;
```

GUIP

Databases for sample application data

Sample application data is stored in several different databases. The default database that is created when DB2 is installed is not used to store the sample application data.

GUIP

DSN8D10P is the database that is used for tables that are related to programs. The other databases are used for tables that are related to applications. The databases are defined by the following statements:

```
CREATE DATABASE DSN8D10A
  STOGROUP DSN8G100
  BUFFERPOOL BP0
  CCSID EBCDIC;

CREATE DATABASE DSN8D10P
  STOGROUP DSN8G100
  BUFFERPOOL BP0
  CCSID EBCDIC;
```

```
CREATE DATABASE DSN8D10L
STOGROUP DSN8G100
BUFFERPOOL BP0
CCSID EBCDIC;
```

```
CREATE DATABASE DSN8D10E
STOGROUP DSN8G100
BUFFERPOOL BP0
CCSID UNICODE;
```

```
CREATE DATABASE DSN8D10U
STOGROUP DSN8G10U
CCSID EBCDIC;
```

GUIP

Table spaces for sample application data

The table spaces that are not explicitly defined are created implicitly in the DSN8D10A database, using the default space attributes.

GUIP

The following SQL statements explicitly define a series of table spaces.

```
CREATE TABLESPACE DSN8S10D
  IN DSN8D10A
  USING STOGROUP DSN8G100
    PRIQTY 20
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S10E
  IN DSN8D10A
  USING STOGROUP DSN8G100
    PRIQTY 20
    SECQTY 20
    ERASE NO
  Numparts 4
    (PART 1 USING STOGROUP DSN8G100
      PRIQTY 12
      SECQTY 12,
     PART 3 USING STOGROUP DSN8G100
      PRIQTY 12
      SECQTY 12)
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  COMPRESS YES
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S10B
  IN DSN8D10L
  USING STOGROUP DSN8G100
    PRIQTY 20
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE
  LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;
```

```

CREATE LOB TABLESPACE DSN8S10M
  IN DSN8D10L
  LOG NO;

CREATE LOB TABLESPACE DSN8S10L
  IN DSN8D10L
  LOG NO;

CREATE LOB TABLESPACE DSN8S10N
  IN DSN8D10L
  LOG NO;

CREATE TABLESPACE DSN8S10C
  IN DSN8D10P
  USING STOGROUP DSN8G100
    PRIQTY 160
    SECQTY 80
  SEGSIZE 4
  LOCKSIZE TABLE
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S10P
  IN DSN8D10A
  USING STOGROUP DSN8G100
    PRIQTY 160
    SECQTY 80
  SEGSIZE 4
  LOCKSIZE ROW
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S10R
  IN DSN8D10A
  USING STOGROUP DSN8G100
    PRIQTY 20
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S10S
  IN DSN8D10A
  USING STOGROUP DSN8G100
    PRIQTY 20
    SECQTY 20
    ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S81Q
  IN DSN8D81P
  USING STOGROUP DSN8G810
    PRIQTY 160
    SECQTY 80
  SEGSIZE 4
  LOCKSIZE PAGE
  BUFFERPOOL BP0
  CLOSE NO
  CCSID EBCDIC;

CREATE TABLESPACE DSN8S81U
  IN DSN8D81E
  USING STOGROUP DSN8G810

```

```
PRIQTY 5
SECQTY 5
ERASE NO
LOCKSIZE PAGE LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
CCSID UNICODE;
```



DB2 sample applications

DB2 provides sample applications to help you with DB2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

This topic describes the DB2 sample applications and the environments under which each application runs. It also provides information on how to use the applications, and how to print the application listings.

You can examine the source code for the sample application programs in the online sample library included with the DB2 product. The name of this sample library is *prefix.SDSNSAMP*.

Using the sample applications

You can use the applications interactively by accessing data in the sample tables on screen displays (panels). You can also access the sample tables in batch when using the phone applications. All sample objects have PUBLIC authorization, which makes the samples easier to run.

Productivity-aid sample programs:

DB2 provides four sample programs that many users find helpful as productivity aids. These programs are shipped as source code, so you can modify them to meet your needs. The programs are:

DSNTIAUL

The sample unload program. This program, which is written in assembler language, is a simple alternative to the UNLOAD utility. It unloads some or all rows from up to 100 DB2 tables. With DSNTIAUL, you can unload data of any DB2 built-in data type or distinct type. DSNTIAUL unloads the rows in a form that is compatible with the LOAD utility and generates utility control statements for LOAD. DSNTIAUL also lets you execute any SQL non-SELECT statement that can be executed dynamically.

DSNTIAD

A sample dynamic SQL program that is written in assembler language. With this program, you can execute any SQL statement that can be executed dynamically, except a SELECT statement.

DSNTEP2

A sample dynamic SQL program that is written in the PL/I language. With this program, you can execute any SQL statement that can be executed dynamically. You can use the source version of DSNTEP2 and modify it to meet your needs, or, if you do not have a PL/I compiler at your installation, you can use the object code version of DSNTEP2.

DSNTEP4

A sample dynamic SQL program that is written in the PL/I language. This program is identical to DSNTEP2 except DSNTEP4 uses multi-row fetch for increased performance. You can use the source version of DSNTEP4 and modify it to meet your needs, or, if you do not have a PL/I compiler at your installation, you can use the object code version of DSNTEP4.

Because these four programs also accept the static SQL statements CONNECT, SET CONNECTION, and RELEASE, you can use the programs to access DB2 tables at remote locations.

Retrieval of UTF-16 Unicode data:

You can use DSNTEP2, DSNTEP4, and DSNTIAUL to retrieve Unicode UTF-16 graphic data. However, these programs might not be able to display some characters, if those characters have no mapping in the target SBCS EBCDIC CCSID.

DSNTIAUL and DSNTIAD are shipped only as source code, so you must precompile, assemble, link, and bind them before you can use them. If you want to use the source code version of DSNTEP2 or DSNTEP4, you must precompile, compile, link, and bind it. You need to bind the object code version of DSNTEP2 or DSNTEP4 before you can use it. Usually a system administrator prepares the programs as part of the installation process. The following table indicates which installation job prepares each sample program. All installation jobs are in data set DSNA10.SDSNSAMP.

Table 187. Jobs that prepare DSNTIAUL, DSNTIAD, DSNTEP2, and DSNTEP4

Program name	Program preparation job
DSNTIAUL	DSNTEJ2A
DSNTIAD	DSNTIJTM
DSNTEP2 (source)	DSNTEJ1P
DSNTEP2 (object)	DSNTEJ1L
DSNTEP4 (source)	DSNTEJ1P
DSNTEP4 (object)	DSNTEJ1L

To run the sample programs, use the DSN RUN command.

The following table lists the load module name and plan name that you must specify, and the parameters that you can specify when you run each program. See the following topics for the meaning of each parameter.

Table 188. DSN RUN option values for DSNTIAUL, DSNTIAD, DSNTEP2, and DSNTEP4

Program name	Load module	Plan	Parameters
DSNTIAUL	DSNTIAUL	DSNTIBA1	SQL number of rows per fetch TOLWARN(NO YES)
DSNTIAD	DSNTIAD	DSNTIAA1	RC0 SQLTERM(termchar)



Table 188. DSN RUN option values for DSNTIAUL, DSNTIAD, DSNTPE2, and DSNTPE4 (continued)

Program name	Load module	Plan	Parameters
DSNTPE2	DSNTPE2	DSNTEPA1	ALIGN(MID) or ALIGN(LHS) NOMIXED or MIXED SQLTERM(<i>termchar</i>) TOLWARN(NO YES) PREPWARN
DSNTPE4	DSNTPE4	DSNTP410	ALIGN(MID) or ALIGN(LHS) NOMIXED or MIXED SQLTERM(<i>termchar</i>) TOLWARN(NO YES) PREPWARN

The remainder of this section contains the following information about running each program:

- Descriptions of the input parameters
- Data sets that you must allocate before you run the program
- Return codes from the program
- Examples of invocation

Related reference:

-  [RUN \(DSN\) \(DB2 Commands\)](#)
-  [DB2 for z/OS Exchange](#)

Types of sample applications

DB2 provides a number of sample applications that manage sample company information. These applications also demonstrate how to use stored procedures, user-defined functions, and LOBs.

Organization application:

The organization application manages the following company information:

- Department administrative structure
- Individual departments
- Individual employees.

Management of information about department administrative structures involves how departments relate to other departments. You can view or change the organizational structure of an individual department, and the information about individual employees in any department. The organization application runs interactively in the ISPF/TSO, IMS, and CICS environments and is available in PL/I and COBOL.

Project application:

The project application manages information about a company's project activities, including the following:

- Project structures
- Project activity listings
- Individual project processing
- Individual project activity estimate processing

- Individual project staffing processing.

Each department works on projects that contain sets of related activities. Information available about these activities includes staffing assignments, completion-time estimates for the project as a whole, and individual activities within a project. The project application runs interactively in IMS and CICS and is available in PL/I only.

Phone application:

The phone application lets you view or update individual employee phone numbers. There are different versions of the application for ISPF/TSO, CICS, IMS, and batch:

- ISPF/TSO applications use COBOL and PL/I.
- CICS and IMS applications use PL/I.
- Batch applications use C, C++, COBOL, FORTRAN, and PL/I.

Stored procedure applications:

There are three sets of stored procedure applications:

IFI applications

These applications let you pass DB2 commands from a client program to a stored procedure, which runs the commands at a DB2 server using the instrumentation facility interface (IFI). There are two sets of client programs and stored procedures. One set has a PL/I client and stored procedure; the other set has a C client and stored procedure.

ODBA application

This application demonstrates how you can use the IMS ODBA interface to access IMS databases from stored procedures. The stored procedure accesses the IMS sample DL/I database. The client program and the stored procedure are written in COBOL.

Utilities stored procedure application

This application demonstrates how to call the utilities stored procedure.

SQL procedure applications

Sample applications are available for both external SQL procedures and native SQL procedures:

- The applications for external SQL procedures demonstrate how to write, prepare, and invoke such procedures. One set of applications demonstrates how to prepare SQL procedures using JCL. The other set of applications shows how to prepare SQL procedures using the SQL procedure processor. The client programs are written in C.
- The sample job for a native SQL procedure shows how to prepare a native SQL procedure, how to manage versions of native SQL procedures, and optionally, how to deploy a native SQL procedure to a remote server. The sample also prepares and executes a sample caller in the C language

WLM refresh application

This application is a client program that calls the DB2-supplied stored procedure WLM_REFRESH to refresh a WLM environment. This program is written in C.

System parameter reporting application

This application is a client program that calls the DB2-supplied stored procedure DSNWZP to display the current settings of system parameters. This program is written in C.

All stored procedure applications run in the TSO batch environment.

User-defined function applications:

The user-defined function applications consist of a client program that invokes the sample user-defined functions and a set of user-defined functions that perform the following functions:

- Convert the current date to a user-specified format
- Convert a date from one format to another
- Convert the current time to a user-specified format
- Convert a date from one format to another
- Return the day of the week for a user-specified date
- Return the month for a user-specified date
- Format a floating point number as a currency value
- Return the table name for a table, view, or alias
- Return the qualifier for a table, view or alias
- Return the location for a table, view or alias
- Return a table of weather information

All programs are written in C or C++ and run in the TSO batch environment.

LOB application:

The LOB application demonstrates how to perform the following tasks:

- Define DB2 objects to hold LOB data
- Populate DB2 tables with LOB data using the LOAD utility, or using INSERT and UPDATE statements when the data is too large for use with the LOAD utility
- Manipulate the LOB data using LOB locators

The programs that create and populate the LOB objects use DSNTIAD and run in the TSO batch environment. The program that manipulates the LOB data is written in C and runs under ISPF/TSO.

Application languages and environments for the sample applications

The sample applications demonstrate how to run DB2 applications in the TSO, IMS, or CICS environments.

The following table shows the environments under which each application runs, and the languages the applications use for each environment.

Table 189. Application languages and environments

Programs	ISPF/TSO	IMS	CICS	Batch	SPUFI
Dynamic SQL programs				Assembler PL/I	

Table 189. Application languages and environments (continued)

Programs	ISPF/TSO	IMS	CICS	Batch	SPUFI
Exit routines	Assembler	Assembler	Assembler	Assembler	Assembler
Organization	COBOL	COBOL PL/I	COBOL PL/I		
Phone	COBOL PL/I Assembler ¹	PL/I	PL/I	COBOL FORTRAN PL/I C C++	
Project		PL/I	PL/I		
SQLCA formatting routines		Assembler	Assembler	Assembler	Assembler
Stored procedures		COBOL		PL/I C SQL	
User-defined functions				C C++	
LOBs	C				

Notes:

1. Assembler subroutine DSN8CA.

Sample applications in TSO

A set of DB2 sample applications run in the TSO environment.

Table 190. Sample DB2 applications for TSO

Application	Program name	Preparation JCL member name	Attachment facility	Description
Phone	DSN8BC3	DSNTEJ2C	DSNELI	This COBOL batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BD3	DSNTEJ2D	DSNELI	This C batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BE3	DSNTEJ2E	DSNELI	This C++ batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BP3	DSNTEJ2P	DSNELI	This PL/I batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BF3	DSNTEJ2F	DSNELI	This FORTRAN program lists employee telephone numbers and updates them if requested.
Organization	DSN8HC3	DSNTEJ3C or DSNTEJ6	DSNALI	This COBOL ISPF program displays and updates information about a local department. It can also display and update information about an employee at a local or remote location.
Phone	DSN8SC3	DSNTEJ3C	DSNALI	This COBOL ISPF program lists employee telephone numbers and updates them if requested.

Table 190. Sample DB2 applications for TSO (continued)

Application	Program name	Preparation JCL member name	Attachment facility	Description
Phone	DSN8SP3	DSNTEJ3P	DSNALI	This PL/I ISPF program lists employee telephone numbers and updates them if requested.
UNLOAD	DSNTIAUL	DSNTEJ2A	DSNELI	This assembler language program unloads the data from a table or view and to produce LOAD utility control statements for the data.
Dynamic SQL	DSNTIAD	DSNTIJTM	DSNELI	This assembler language program dynamically executes non-SELECT statements read in from SYSIN; that is, it uses dynamic SQL to execute non-SELECT SQL statements.
Dynamic SQL	DSNTEP2	DSNTEJ1P or DSNTEJ1L	DSNELI	This PL/I program dynamically executes SQL statements read in from SYSIN. Unlike DSNTIAD, this application can also execute SELECT statements.
Stored procedures ¹	DSN8EP1	DSNTEJ6P	DSNELI	The jobs DSNTEJ6P and DSNTEJ6S prepare a PL/I version of the application. This sample executes DB2 commands using the instrumentation facility interface (IFI).
Stored procedure ¹	DSN8EP2	DSNTEJ6S	DSNRLI	
Stored procedures ¹	DSN8EPU	DSNTEJ6U	DSNELI	The sample that is prepared by job DSNTEJ6U invokes the utilities stored procedure.
Stored procedures ¹	DSN8ED1	DSNTEJ6D	DSNELI	The jobs DSNTEJ6D and DSNTEJ6T prepare a C version of the application. The C stored procedure uses result sets to return commands to the client. This sample executes DB2 commands using the instrumentation facility interface (IFI).
Stored procedures ¹	DSN8ED2	DSNTEJ6T	DSNRLI	
Stored procedures ¹	DSN8EC1	DSNTEJ61	DSNRLI	The sample that is prepared by jobs DSNTEJ61 and DSNTEJ62 demonstrates a stored procedure that accesses IMS databases through the ODBA interface.
Stored procedures ¹	DSN8EC2	DSNTEJ62	DSNELI	
Stored procedures ¹	DSN8ES1	DSNTEJ63	DSNRLI	The sample that is prepared by jobs DSNTEJ63 and DSNTEJ64 demonstrates how to prepare an SQL procedure using JCL.
Stored procedures ¹	DSN8ED3	DSNTEJ64	DSNELI	
Stored procedures ¹	DSN8ES2	DSNTEJ65	DSNRLI	The sample that is prepared by job DSNTEJ65 demonstrates how to prepare an SQL procedure using the SQL procedure processor.
Stored procedures ¹	DSN8ED6	DSNTEJ6W	DSNELI	The sample that is prepared by job DSNTEJ6W demonstrates how to prepare and run a client program that calls a DB2-supplied stored procedure to refresh a WLM environment.
Stored procedures ¹	DSN8ED7	DSNTEJ6Z	DSNELI	The sample that is prepared by job DSNTEJ6Z demonstrates how to prepare and run a client program that calls a DB2-supplied stored procedure to display the current settings of system parameters.

Table 190. Sample DB2 applications for TSO (continued)

Application	Program name	Preparation JCL member name	Attachment facility	Description
Stored procedures ¹	DSN8ED9	DSNTEJ66	DSNELI	The sample that is prepared by job DSNTEJ66 demonstrates how to prepare and run a client program that calls a native SQL procedure, manages versions of that procedure, and optionally, deploys that procedure to a remote server. DSN8ES3 is the sample native SQL procedure and DSN8ED9 is the sample C language caller of DSN8ES3.
Stored procedures ¹	DSN8ES3	DSNTEJ66	not applicable	
User-defined functions	DSN8DUAD	DSNTEJ2U	DSNRLI	These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.
User-defined functions	DSN8DUAT	DSNTEJ2U	DSNRLI	
User-defined functions	DSN8DUCD	DSNTEJ2U	DSNRLI	
User-defined functions	DSN8DUCT	DSNTEJ2U	DSNRLI	
User-defined functions	DSN8DUCY	DSNTEJ2U	DSNRLI	
User-defined functions	DSN8DUTI	DSNTEJ2U	DSNRLI	
User-defined functions	DSN8DUWC	DSNTEJ2U	DSNRLI	
User-defined functions	DSN8DUWF	DSNTEJ2U	DSNRLI	The user-defined table function DSN8DUWF can be invoked by the C client program DSN8DUWC.
User-defined functions	DSN8EUDN	DSNTEJ2U	DSNRLI	These C++ applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.
User-defined functions	DSN8EUMN	DSNTEJ2U	DSNRLI	
LOBs	DSN8DLPL	DSNTEJ71	DSNELI	These applications demonstrate how to populate a LOB column that is greater than 32 KB, manipulate the data using the POSSTR and SUBSTR built-in functions, and display the data in ISPF using GDDM [®] .
LOBs	DSN8DLCT	DSNTEJ71	DSNELI	
LOBs	DSN8DLRV	DSNTEJ73	DSNELI	
LOBs	DSN8DLPV	DSNTEJ75	DSNELI	

Note:

1. All of the stored procedure applications consist of a calling program, a stored procedure program, or both.

Related reference:

“Data sets that the precompiler uses” on page 921

Sample applications in IMS

A set of DB2 sample applications run in the IMS environment.

Table 191. Sample DB2 applications for IMS

Application	Program name	JCL member name	Description
Organization	DSN8IC0 DSN8IC1 DSN8IC2	DSNTEJ4C	IMS COBOL Organization Application
Organization	DSN8IP0 DSN8IP1 DSN8IP2	DSNTEJ4P	IMS PL/I Organization Application
Project	DSN8IP6 DSN8IP7 DSN8IP8	DSNTEJ4P	IMS PL/I Project Application
Phone	DSN8IP3	DSNTEJ4P	IMS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

Related reference:

“Data sets that the precompiler uses” on page 921

Sample applications in CICS

A set of DB2 sample applications run in the CICS environment.

Table 192. Sample DB2 applications for CICS

Application	Program name	JCL member name	Description
Organization	DSN8CC0 DSN8CC1 DSN8CC2	DSNTEJ5C	CICS COBOL Organization Application
Organization	DSN8CP0 DSN8CP1 DSN8CP2	DSNTEJ5P	CICS PL/I Organization Application
Project	DSN8CP6 DSN8CP7 DSN8CP8	DSNTEJ5P	CICS PL/I Project Application
Phone	DSN8CP3	DSNTEJ5P	CICS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

Related reference:

“Data sets that the precompiler uses” on page 921

DSNTIAUL

Use the DSNTIAUL program to unload data from DB2 tables into sequential data sets. The data is copied to the data sets and is not deleted from the table.

When multi-row fetch is used, parallelism might be disabled in the last parallel group in the top-level query block for a query. For very simple queries, parallelism might be disabled for the entire query when multi-row fetch is used. To obtain full parallelism when running DSNTIAUL, switch DSNTIAUL to single-row fetch mode by specifying 1 for the number of rows per fetch parameter.

DSNTIAUL uses SQL to access DB2. Operations on a row-level or column-level access control enforced table are subject to the rules specified for the access control. If the table is row-level access control enforced, DSNTIAUL receives and returns only the rows of the table that satisfy the row permissions for the user. If the table is column-level access control enforced, DSNTIAUL receives and returns the values in the column values as modified by the column masks for the user.

DSNTIAUL parameters:

SQL

Specify SQL to indicate that your input data set contains one or more complete SQL statements, each of which ends with a semicolon. You can include any SQL statement that can be executed dynamically in your input data set. In addition, you can include the static SQL statements CONNECT, SET CONNECTION, or RELEASE. Static SQL statements must be uppercase.

DSNTIAUL uses the SELECT statements to determine which tables to unload and dynamically executes all other statements except CONNECT, SET CONNECTION, and RELEASE. DSNTIAUL executes CONNECT, SET CONNECTION, and RELEASE statically to connect to remote locations.

number of rows per fetch

Specify a number from 1 to 32767 to specify the number of rows per fetch that DSNTIAUL retrieves. If you do not specify this number, DSNTIAUL retrieves 100 rows per fetch. This parameter can be specified with the SQL parameter.

TOLWARN

Specify NO (the default) or YES to indicate whether DSNTIAUL continues to retrieve rows after receiving an SQL warning:

(NO) If a warning occurs when DSNTIAUL executes an OPEN or FETCH to retrieve rows, DSNTIAUL stops retrieving rows. If the SQLWARN1, SQLWARN2, SQLWARN6, or SQLWARN7 flag is set when DSNTIAUL executes a FETCH to retrieve rows, DSNTIAUL continues to retrieve rows.

(YES) If a warning occurs when DSNTIAUL executes an OPEN or FETCH to retrieve rows, DSNTIAUL continues to retrieve rows.

LOBFIL(*prefix*)

Specify LOBFIL to indicate that you want DSNTIAUL to dynamically allocate data sets, each to receive the full content of a LOB cell. (A LOB cell is the intersection of a row and a LOB column.) If you do not specify the LOBFIL option, you can unload up to only 32 KB of data from a LOB column.

prefix

Specify a high-level qualifier for these dynamically allocated data sets. You can specify up to 17 characters. The qualifier must conform with the rules for TSO data set names.

DSNTIAUL uses a naming convention for these dynamically allocated data sets of *prefix.Qiiiiiii.Cjjjjjj.Rkkkkkkk*, where these qualifiers have the following values:

prefix

The high-level qualifier that you specify in the LOBFILE option.

Qiiiiiii

The sequence number (starting from 0) of a query that returns one or more LOB columns

Cjjjjjjj

The sequence number (starting from 0) of a column in a query that returns one or more LOB columns

Rkkkkkkk

The sequence number (starting from 0) of a row of a result set that has one or more LOB columns.

The generated LOAD statement contains LOB file reference variables that can be used to load data from these dynamically allocated data sets.

If you do not specify the SQL parameter, your input data set must contain one or more single-line statements (without a semicolon) that use the following syntax:

table or view name [WHERE conditions] [ORDER BY columns]

Each input statement must be a valid SQL SELECT statement with the clause SELECT * FROM omitted and with no ending semicolon. DSNTIAUL generates a SELECT statement for each input statement by appending your input line to SELECT * FROM, then uses the result to determine which tables to unload. For this input format, the text for each table specification can be a maximum of 72 bytes and must not span multiple lines.

You can use the input statements to specify SELECT statements that join two or more tables or select specific columns from a table. If you specify columns, you need to modify the LOAD statement that DSNTIAUL generates.

DSNTIAUL data sets:

Data set

Description

SYSIN

Input data set.

You cannot enter comments in DSNTIAUL input.

The record length for the input data set must be at least 72 bytes. DSNTIAUL reads only the first 72 bytes of each record.

SYSPRINT

Output data set. DSNTIAUL writes informational and error messages in this data set.

The record length for the SYSPRINT data set is 121 bytes.

SYSPUNCH

Output data set. DSNTIAUL writes the LOAD utility control statements in this data set.

SYSREC_{nn}

Output data sets. The value *nn* ranges from 00 to 99. You can have a maximum of 100 output data sets for a single execution of DSNTIAUL. Each data set contains the data that is unloaded when DSNTIAUL processes a SELECT statement from the input data set. Therefore, the

number of output data sets must match the number of SELECT statements (if you specify parameter SQL) or table specifications in your input data set.

Define all data sets as sequential data sets. You can specify the record length and block size of the SYSPUNCH and SYSRECCnn data sets. The maximum record length for the SYSPUNCH and SYSRECCnn data sets is 32760 bytes.

DSNTIAUL return codes:

Table 193. DSNTIAUL return codes

Return code	Meaning
0	Successful completion.
4	An SQL statement received a warning code. <ul style="list-style-type: none"> • If TOLWARN(YES) is specified, and the warning occurred on a FETCH or OPEN during the processing of a SELECT statement, DB2 performs the unload operation. • Otherwise if the SQL statement was a SELECT statement, DB2 did not perform the associated unload operation. <p>If DB2 returns a +394, which indicates that it is using optimization hints, or a +395, which indicates one or more invalid optimization hints, DB2 performs the unload operation.</p>
8	An SQL statement received an error code. If the SQL statement was a SELECT statement, DB2 did not perform the associated unload operation or did not complete it.
12	DSNTIAUL could not open a data set, an SQL statement returned a severe error code (-144, -302, -804, -805, -818, -902, -906, -911, -913, -922, -923, -924, or -927), or an error occurred in the SQL message formatting routine.

Example of using DSNTIAUL to unload a subset of rows in a table:

Suppose that you want to unload the rows for department D01 from the project table. Because you can fit the table specification on one line, and you do not want to execute any non-SELECT statements, you do not need the SQL parameter. Your invocation looks like the one that is shown in the following figure:

```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBA1) -
LIB('DSNA10.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSRECO0 DD DSN=DSN8UNLD.SYSRECO0,
// UNIT=SYSDA,SPACE=(32760,(1000,500)),DISP=(,CATLG),
// VOL=SER=SCR03
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
// UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
// VOL=SER=SCR03,RECFM=FB,LRECL=120,BLKSIZE=1200
//SYSIN DD *DSN8A10
.PROJ WHERE DEPTNO='D01'
```

Example of using DSNTIAUL to unload rows in more than one table:

Suppose that you also want to use DSNTIAUL to perform the following actions:

- Unload all rows from the project table
- Unload only rows from the employee table for employees in departments with department numbers that begin with D, and order the unloaded rows by employee number
- Lock both tables in share mode before you unload them
- Retrieve 250 rows per fetch

For these activities, you must specify the SQL parameter and specify the number of rows per fetch when you run DSNTIAUL. Your DSNTIAUL invocation is shown in the following figure:

```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBA1) PARS('SQL,250') -
LIB('DSNA10.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=DSN8UNLD.SYSREC00,
// UNIT=SYSDA,SPACE=(32760,(1000,500)),DISP=(,CATLG),
// VOL=SER=SCR03
//SYSREC01 DD DSN=DSN8UNLD.SYSREC01,
// UNIT=SYSDA,SPACE=(32760,(1000,500)),DISP=(,CATLG),
// VOL=SER=SCR03
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
// UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
// VOL=SER=SCR03,RECFM=FB,LRECL=120,BLKSIZE=1200
//SYSIN DD *
LOCK TABLE DSN8A10.EMP IN SHARE MODE;
LOCK TABLE DSN8A10.PROJ IN SHARE MODE;
SELECT * FROM DSN8A10.PROJ;
SELECT * FROM DSN8A10.EMP
WHERE WORKDEPT LIKE 'D%'
ORDER BY EMPNO;
```

Example of using DSNTIAUL to obtain LOAD utility control statements:

If you want to obtain the LOAD utility control statements for loading rows into a table, but you do not want to unload the rows, you can set the data set names for the SYSREC nm data sets to DUMMY. For example, to obtain the utility control statements for loading rows into the department table, you invoke DSNTIAUL as shown in the following figure:

```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBA1) -
LIB('DSNA10.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DUMMY
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
// UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
// VOL=SER=SCR03,RECFM=FB,LRECL=120,BLKSIZE=1200
//SYSIN DD *DSN8A10
.DEPT
```

Example of using DSNTIAUL to unload LOB data:

This example uses the sample LOB table with the following structure:

```
CREATE TABLE DSN8910.EMP_PHOTO_RESUME
( EMPNO CHAR(06) NOT NULL,
EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,
PSEG_PHOTO BLOB(500K),
BMP_PHOTO BLOB(100K),
RESUME CLOB(5K),
PRIMARY KEY (EMPNO))
IN DSN8D91L.DSN8S91B
CCSID EBCDIC;
```

The following call to DSNTIAUL unloads the sample LOB table. The parameters for DSNTIAUL indicate the following options:

- The input data set (SYSIN) contains SQL.
- DSNTIAUL is to retrieve 2 rows per fetch.
- DSNTIAUL places the LOB data in data sets with a high-level qualifier of DSN8UNLD.

```
//UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB91) -
PARMS('SQL,2,LOBFILE(DSN8UNLD)') -
LIB('DSN910.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSRECO0 DD DSN=DSN8UNLD.SYSRECO0,
// UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
// VOL=SER=SCR03,RECFM=FB
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
// UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
// VOL=SER=SCR03,RECFM=FB
//SYSIN DD *
SELECT * FROM DSN8910.EMP_PHOTO_RESUME;
```

Given that the sample LOB table has 4 rows of data, DSNTIAUL produces the following output:

- Data for columns EMPNO and EMP_ROWID are placed in the data set that is allocated according to the SYSRECO0 DD statement. The data set name is DSN8UNLD.SYSRECO0
- A generated LOAD statement is placed in the data set that is allocated according to the SYSPUNCH DD statement. The data set name is DSN8UNLD.SYSPUNCH
- The following data sets are dynamically created to store LOB data:
 - DSN8UNLD.Q0000000.C0000002.R0000000
 - DSN8UNLD.Q0000000.C0000002.R0000001
 - DSN8UNLD.Q0000000.C0000002.R0000002
 - DSN8UNLD.Q0000000.C0000002.R0000003
 - DSN8UNLD.Q0000000.C0000003.R0000000
 - DSN8UNLD.Q0000000.C0000003.R0000001
 - DSN8UNLD.Q0000000.C0000003.R0000002
 - DSN8UNLD.Q0000000.C0000003.R0000003
 - DSN8UNLD.Q0000000.C0000004.R0000000
 - DSN8UNLD.Q0000000.C0000004.R0000001

- DSN8UNLD.Q0000000.C0000004.R0000002
- DSN8UNLD.Q0000000.C0000004.R0000003

For example, DSN8UNLD.Q0000000.C0000004.R0000001 means that the data set contains data that is unloaded from the second row (R0000001) and the fifth column (C0000004) of the result set for the first query (Q0000000).

DSNTIAD

Use the DSNTIAD program to execute SQL statements other than SELECT statements dynamically.

DSNTIAD parameters:

RC0

If you specify this parameter, DSNTIAD ends with return code 0, even if the program encounters SQL errors. If you do not specify RC0, DSNTIAD ends with a return code that reflects the severity of the errors that occur. Without RC0, DSNTIAD terminates if more than 10 SQL errors occur during a single execution.

SQLTERM(*termchar*)

Specify this parameter to indicate the character that you use to end each SQL statement. You can use any special character **except** one of those listed in the following table. SQLTERM(;) is the default.

Table 194. Invalid special characters for the SQL terminator

Name	Character	Hexadecimal representation
blank		X'40'
comma	,	X'6B'
double quotation mark	"	X'7F'
left parenthesis	(X'4D'
right parenthesis)	X'5D'
single quotation mark	'	X'7D'
underscore	_	X'6D'

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons.

example:

Suppose that you specify the parameter SQLTERM(#) to indicate that the character # is the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

A CREATE PROCEDURE statement with embedded semicolons looks like the following statement:

```
CREATE PROCEDURE PROC1 (IN PARM1 INT, OUT SCODE INT)
  LANGUAGE SQL
  BEGIN
```

```

DECLARE SQLCODE INT;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
    SET SCODE = SQLCODE;
UPDATE TBL1 SET COL1 = PARM1;
END #

```

Be careful to choose a character for the statement terminator that is not used within the statement.

DSNTIAD data sets:

Data set

Description

SYSIN

Input data set. In this data set, you can enter any number of non-SELECT SQL statements, each terminated with a semicolon. A statement can span multiple lines, but DSNTIAD reads only the first 72 bytes of each line.

You cannot enter comments in DSNTIAD input.

SYSPRINT

Output data set. DSNTIAD writes informational and error messages in this data set. DSNTIAD sets the record length of this data set to 121 bytes and the block size to 1210 bytes.

Define all data sets as sequential data sets.

DSNTIAD return codes:

Table 195. DSNTIAD return codes

Return code	Meaning
0	Successful completion, or the user-specified parameter RC0.
4	An SQL statement received a warning code.
8	An SQL statement received an error code.
12	DSNTIAD could not open a data set, the length of an SQL statement was more than 2 MB, an SQL statement returned a severe error code (-8nn or -9nn), or an error occurred in the SQL message formatting routine.

Example of DSNTIAD invocation:

Suppose that you want to execute 20 UPDATE statements, and you do not want DSNTIAD to terminate if more than 10 errors occur. Your invocation looks like the one that is shown in the following figure:

```

//RUNTIAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSPRINT DD SYSOUT=*
//SYSTSIN DD *
  DSN SYSTEM(DSN)
  RUN PROGRAM(DSNTIAD) PLAN(DSNTIAA1) PARM('RC0') -
    LIB('DSNA10.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
UPDATE DSN8A10.PROJ SET DEPTNO='J01' WHERE DEPTNO='A01';
UPDATE DSN8A10.PROJ SET DEPTNO='J02' WHERE DEPTNO='A02';
:
UPDATE DSN8A10.PROJ SET DEPTNO='J20' WHERE DEPTNO='A20';

```

DSNTEP2 and DSNTEP4

Use the DSNTEP2 or DSNTEP4 programs to execute SQL statements dynamically.

DSNTEP4 is identical to DSNTEP2 except that it uses multi-row fetch for increased performance. When multi-row fetch is used, parallelism might be disabled in the last parallel group in the top-level query block for a query. For very simple queries, parallelism might be disabled for the entire query when multi-row fetch is used. To obtain full parallelism, either use DSNTEP2 or specify the control option SET MULT_FETCH 1 for DSNTEP4.

DSNTEP2 and DSNTEP4 write their results to the data set that is defined by the SYSPRINT DD statement. SYSPRINT data must have a logical record length of 133 bytes (LRECL=133). Otherwise, the program issues return code 12 with abend U4038 and reason code 1. This abend occurs due to the PL/I file exception error IBM0201S ONCODE=81. The following error message is issued:

The UNDEFINEDFILE condition was raised because of conflicting DECLARE and OPEN attributes (FILE= SYSPRINT).

If you use applications or other automation to process output from DSNTEP2 or DSNTEP4, be aware that minor changes in the format can occur as a result of service or enhancements. Such changes might require you to adjust your processes that use the output of these programs.

Important: When you allocate a new data set with the SYSPRINT DD statement, either specify a DCB with RECFM=FBA and LRECL=133, or do not specify the DCB parameter.

DSNTEP2 and DSNTEP4 parameters:

ALIGN(MID) or ALIGN(LHS)

Specifies the alignment.

ALIGN(MID)

Specifies that DSNTEP2 or DSNTEP4 output should be centered.

ALIGN(MID) is the default.

ALIGN(LHS)

Specifies that the DSNTEP2 or DSNTEP4 output should be left-justified.

NOMIXED or MIXED

Specifies whether DSNTEP2 or DSNTEP4 contains any DBCS characters.

NOMIXED

Specifies that the DSNTEP2 or DSNTEP4 input contains no DBCS characters. NOMIXED is the default.

MIXED

Specifies that the DSNTEP2 or DSNTEP4 input contains some DBCS characters.

PREPWARN

Specifies that DSNTEP2 or DSNTEP4 is to display details about any SQL warnings that are encountered at PREPARE time.

Regardless of whether you specify PREPWARN, when an SQL warning is encountered at PREPARE time, the program displays the message SQLWARNING ON PREPARE and sets the return code to 4. When you specify PREPWARN, the program also displays the details about any SQL warnings.

SQLFORMAT

Specifies how DSNTEP2 or DSNTEP4 pre-processes SQL statements before passing them to DB2. Select one of the following options:

SQL This is the preferred mode for SQL statements other than SQL procedural language. When you use this option, which is the default, DSNTEP2 or DSNTEP4 collapses each line of an SQL statement into a single line before passing the statement to DB2. DSNTEP2 or DSNTEP4 also discards all SQL comments.

SQLCOMNT

This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, behavior is similar to SQL mode, except that DSNTEP2 or DSNTEP4 does not discard SQL comments. Instead, it automatically terminates each SQL comment with a line feed character (hex 25), unless the comment is already terminated by one or more line formatting characters. Use this option to process SQL procedural language with minimal modification by DSNTEP2 or DSNTEP4.

SQLPL

This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, DSNTEP2 or DSNTEP4 retains SQL comments and terminates each line of an SQL statement with a line feed character (hex 25) before passing the statement to DB2. Lines that end with a split token are not terminated with a line feed character. Use this mode to obtain improved diagnostics and debugging of SQL procedural language.

SQLTERM(*termchar*)

Specifies the character that you use to end each SQL statement. You can use any character except one of those that are listed in Table 194 on page 1106.

SQLTERM(;) is the default.

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons.

Example: Suppose that you specify the parameter SQLTERM(#) to indicate that the character # is the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

A CREATE PROCEDURE statement with embedded semicolons looks like the following statement:

```
CREATE PROCEDURE PROC1 (IN PARM1 INT, OUT SCODE INT)
  LANGUAGE SQL
  BEGIN
    DECLARE SQLCODE INT;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
      SET SCODE = SQLCODE;
    UPDATE TBL1 SET COL1 = PARM1;
  END #
```

Be careful to choose a character for the statement terminator that is not used within the statement.

If you want to change the SQL terminator within a series of SQL statements, you can use the `--#SET TERMINATOR` control statement.

Example: Suppose that you have an existing set of SQL statements to which you want to add a `CREATE TRIGGER` statement that has embedded semicolons. You can use the default `SQLTERM` value, which is a semicolon, for all of the existing SQL statements. Before you execute the `CREATE TRIGGER` statement, include the `--#SET TERMINATOR #` control statement to change the SQL terminator to the character `#`:

```
SELECT * FROM DEPT;
SELECT * FROM ACT;
SELECT * FROM EMPPROJECT;
SELECT * FROM PROJ;
SELECT * FROM PROJECT;
--#SET TERMINATOR #
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMP
  FOR EACH ROW MODE DB2SQL
  BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
  END#
```

See the following discussion of the `SYSIN` data set for more information about the `--#SET` control statement.

TOLWARN

Indicates whether `DSNTEP2` or `DSNTEP4` continues to process SQL `SELECT` statements after receiving an SQL warning. You can specify one of the following values:

NO Indicates that the program stops processing the `SELECT` statement if a warning occurs when the program executes an `OPEN` or `FETCH` for a `SELECT` statement. `NO` is the default value for `TOLWARN`.

The following exceptions exist:

- If `SQLCODE +445` or `SQLCODE +595` occurs when `DSNTEP2` or `DSNTEP4` executes a `FETCH` for a `SELECT` statement, the program continues to process the `SELECT` statement.
- If `SQLCODE +354` occurs when `DSNTEP4` executes a `FETCH` for a `SELECT` statement, the program continues to process the `SELECT` statement.
- If `SQLCODE +802` occurs when `DSNTEP2` or `DSNTEP4` executes a `FETCH` for a `SELECT` statement, the program continues to process the `SELECT` statement if the `TOLARTHWRN` control statement is set to `YES`.

YES

Indicates that the program continues to process the `SELECT` statement if a warning occurs when the program executes an `OPEN` or `FETCH` for a `SELECT` statement.

DSNTEP2 and DSNTEP4 data sets:

The following data sets are used by `DSNTEP2` and `DSNTEP4`:

SYSIN

Input data set. In this data set, you can enter any number of SQL statements, each terminated with a semicolon. A statement can span multiple lines, but `DSNTEP2` or `DSNTEP4` reads only the first 72 bytes of each line. You must explicitly commit any SQL statements except the last one.

You can enter comments in DSNTEP2 or DSNTEP4 input with an asterisk (*) in column 1 or two hyphens (--) anywhere on a line. Text that follows the asterisk is considered to be comment text. Text that follows two hyphens can be comment text or a control statement. Comments are not considered in dynamic statement caching. Comments and control statements cannot span lines.

You can enter control statements of the following form in the DSNTEP2 and DSNTEP4 input data set:

```
--#SET control-option value
```

You can specify the following control options. If you specify a value of NO for any of the options in this list, the program behaves as if you did not specify the parameter.

TERMINATOR

The SQL statement terminator. *value* is any single-byte character other than one of those that are listed in Table 194 on page 1106. The default is the value of the SQLTERM parameter.

ROWS_FETCH

The number of rows that are to be fetched from the result table. *value* is a numeric literal between -1 and the number of rows in the result table. -1 means that all rows are to be fetched. The default is -1.

ROWS_OUT

The number of fetched rows that are to be sent to the output data set. *value* is a numeric literal between -1 and the number of fetched rows. -1 means that all fetched rows are to be sent to the output data set. The default is -1.

MULT_FETCH

This option is valid only for DSNTEP4. Use MULT_FETCH to specify the number of rows that are to be fetched at one time from the result table. The default fetch amount for DSNTEP4 is 100 rows, but you can specify from 1 to 32676 rows.

TOLWARN

Indicates whether DSNTEP2 or DSNTEP4 continues to process SQL SELECT statements after receiving an SQL warning. You can specify one of the following values:

NO Indicates that the program stops processing the SELECT statement if a warning occurs when the program executes an OPEN or FETCH for a SELECT statement. NO is the default value for TOLWARN.

The following exceptions exist:

- If SQLCODE +445 or SQLCODE +595 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement.
- If SQLCODE +354 occurs when DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement.
- If SQLCODE +802 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement if the TOLARTHWRN control statement is set to YES.

|
| **YES**

| Indicates that the program continues to process the SELECT
| statement if a warning occurs when the program executes an
| OPEN or FETCH for a SELECT statement.

| **TOLARTHWRN**

| Indicates whether DSNTEP2 and DSNTEP4 continue to process an SQL
| SELECT statement after an arithmetic SQL warning (SQLCODE +802)
| is returned. *value* is either NO (the default) or YES.

| **PREPWARN**

| Specifies that DSNTEP2 or DSNTEP4 is to display details about any
| SQL warnings that are encountered at PREPARE time.

| Regardless of whether you specify PREPWARN, when an SQL warning
| is encountered at PREPARE time, the program displays the message
| SQLWARNING ON PREPARE and sets the return code to 4. When you
| specify PREPWARN, the program also displays the details about any
| SQL warnings.

| **SQLFORMAT**

| Specifies how DSNTEP2 or DSNTEP4 pre-processes SQL statements
| before passing them to DB2. Select one of the following options:

| **SQL** This is the preferred mode for SQL statements other than SQL
| procedural language. When you use this option, which is the
| default, DSNTEP2 or DSNTEP4 collapses each line of an SQL
| statement into a single line before passing the statement to
| DB2. DSNTEP2 or DSNTEP4 also discards all SQL comments.

| **SQLCOMNT**

| This mode is suitable for all SQL, but it is intended primarily
| for SQL procedural language processing. When this option is in
| effect, behavior is similar to SQL mode, except that DSNTEP2
| or DSNTEP4 does not discard SQL comments. Instead, it
| automatically terminates each SQL comment with a line feed
| character (hex 25), unless the comment is already terminated
| by one or more line formatting characters. Use this option to
| process SQL procedural language with minimal modification
| by DSNTEP2 or DSNTEP4.

| **SQLPL**

| This mode is suitable for all SQL, but it is intended primarily
| for SQL procedural language processing. When this option is in
| effect, DSNTEP2 or DSNTEP4 retains SQL comments and
| terminates each line of an SQL statement with a line feed
| character (hex 25) before passing the statement to DB2. Lines
| that end with a split token are not terminated with a line feed
| character. Use this mode to obtain improved diagnostics and
| debugging of SQL procedural language.

| **MAXERRORS**

| Specifies that number of errors that DSNTEP2 and DSNTEP4 handle
| before processing stops. The default is 10.

| **SYSPRINT**

| Output data set. DSNTEP2 and DSNTEP4 write informational and error
| messages in this data set. DSNTEP2 and DSNTEP4 write output records of
| no more than 133 bytes.

Define all data sets as sequential data sets.

DSNTEP2 and DSNTEP4 return codes

Table 196. DSNTEP2 and DSNTEP4 return codes

Return code	Meaning
0	Successful completion.
4	An SQL statement received a warning code.
8	An SQL statement received an error code.
12	The length of an SQL statement was more than 32760 bytes, an SQL statement returned a severe error code (-8nn or -9nn), or an error occurred in the SQL message formatting routine.

Example of DSNTEP2 invocation

Suppose that you want to use DSNTEP2 to execute SQL SELECT statements that might contain DBCS characters. You also want left-aligned output. Your invocation looks like the one in the following figure.

```
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
  DSN SYSTEM(DSN)
  RUN PROGRAM(DSNTEP2) PLAN(DSNTEPA1) PARS('/ALIGN(LHS) MIXED TOLWARN(YES)') -
    LIB('DSNA10.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
SELECT * FROM DSN8A10.PROJ;
```

Example of DSNTEP4 invocation

Suppose that you want to use DSNTEP4 to execute SQL SELECT statements that might contain DBCS characters, and you want center-aligned output. You also want DSNTEP4 to fetch 250 rows at a time. Your invocation looks like the one in the following figure:

```
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
  DSN SYSTEM(DSN)
  RUN PROGRAM(DSNTEP4) PLAN(DSNTEPA1) PARS('/ALIGN(MID) MIXED') -
    LIB('DSNA10.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
--#SET MULT_FETCH 250
SELECT * FROM DSN8A10.EMP;
```

Information resources for DB2 for z/OS and related products

Information about DB2 for z/OS and products that you might use in conjunction with DB2 for z/OS is available in online information centers or on library websites.

Obtaining DB2 for z/OS publications

The current DB2 for z/OS publications are available from the following website:

<http://www-01.ibm.com/support/docview.wss?uid=swg27019288>

Links to the information center version and the PDF version of each publication are provided.

DB2 for z/OS publications are also available for download from the IBM Publications Center (<http://www.ibm.com/shop/publications/order>).

In addition, books for DB2 for z/OS are available on a CD-ROM that is included with your product shipment:

- DB2 10 for z/OS Licensed Library Collection, LK5T-7390, in English. The CD-ROM contains the collection of books for DB2 10 for z/OS in PDF format. Periodically, IBM refreshes the books on subsequent editions of this CD-ROM.

Installable information center

You can download or order an installable version of the Information Management Software for z/OS Solutions Information Center, which includes information about DB2 for z/OS, QMF, IMS, and many DB2 and IMS Tools products. You can install this information center on a local system or on an intranet server. For more information, see http://www-01.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/com.ibm.db2z11.doc/src/alltoc/installabledzic.html.

Notices

This information was developed for products and services offered in the U.S.A. This material may be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This information is intended to help you to write programs that contain SQL statements. This information primarily documents General-use Programming Interface and Associated Guidance Information provided by DB2 10 for z/OS. However, this information also documents Product-sensitive Programming Interface and Associated Guidance Information provided by DB2 10 for z/OS.


General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 10 for z/OS.

Product-sensitive Programming Interface and Associated Guidance Information

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

 Product-sensitive Programming Interface and Associated Guidance Information... 

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com)[®] are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at <http://www.ibm.com/legal/copytrade.shtml>.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

Glossary

The glossary is available in the Information Management Software for z/OS Solutions Information Center.

See the Glossary topic for definitions of DB2 for z/OS terms.

Index

Special characters

- _ (underscore)
 - assembler host variable 267
- ' (apostrophe)
 - string delimiter precompiler option 933

Numerics

- 31-bit addressing 1005

A

- abend
 - effect on cursor position 725
 - for synchronization calls 772
 - IMS
 - U0102 1025
 - system
 - X"04E" 772
- abend recovery routine
 - in CAF 79
- access path
 - direct row access 754
- accessibility
 - keyboard xvi
 - shortcut keys xvi
- accessing data
 - from an application program 677
- activity sample table 1071
- adding
 - data 655
- ALL quantified predicate 712
- ALTER PROCEDURE statement
 - external stored procedure 653
- AMODE link-edit option 942, 1005
- ANY quantified predicate 712
- APOST precompiler option 933
- application
 - rebinding 962
 - application 962
- application plan
 - binding 949
 - creating 943
 - dynamic plan selection for CICS applications 961
 - listing packages 949
 - rebinding 964
- application program
 - bill of materials 489
 - checking success of SQL statements 173
 - coding SQL statements 159
 - coding conventions 251
 - data entry 655
 - dynamic SQL 193, 198
 - selecting rows using a cursor 722
 - design considerations
 - checkpoint 772
 - IMS calls 772
 - programming for DL/I batch 772
 - SQL statements 772
 - structure 1021
 - application program (*continued*)
 - design considerations (*continued*)
 - synchronization call abends 772
 - using ISPF (interactive system productivity facility) 915
 - XRST call 772
 - duplicate CALL statements 803
 - external stored procedures 569
 - including queries 159
 - object extensions 512
 - preparation
 - assembling 942
 - binding 943
 - compiling 942
 - DB2 precompiler option defaults 940
 - defining to CICS 942
 - DRDA access 954
 - example 983
 - link-editing 942
 - precompiler option defaults 918
 - preparing for running 915
 - program preparation panel 915
 - using DB2I (DB2 Interactive) 915
 - preparation overview 973
 - running
 - CICS 1030
 - IMS 1030
 - program synchronization in DL/I batch 772
 - TSO 1017
 - TSO CLIST 1030
 - table and view declarations 161
 - test environment 1017
 - testing 1017
- application program design
 - planning for changes 52
- application programming
 - DCLGEN example 170
 - DCLGEN variable declarations 167
- application programs
 - compatible data types 180
 - host structures 175
 - host variable arrays 175
 - host variables 174
 - performance 772
- applications
 - designing 1
 - migrating 1
 - planning 1
- arithmetic expressions in UPDATE statement 670
- array pointer host variable
 - declaring 300
 - referencing in SQL statements 298
- AS clause
 - naming columns for view 683
 - naming columns in union 683
 - naming derived columns 683
 - naming result columns 683
 - ORDER BY name 681
- ASCII data, retrieving 202
- assembler application program
 - assembling 942

- assembler application program (*continued*)
 - data type compatibility 261
 - declaring tables 267
 - declaring views 267
 - defining the SQLDA 173, 254
 - host variable
 - naming convention 267
 - host variable, declaring 254
 - INCLUDE statement 267
 - including SQLCA 253
 - indicator variable declaration 260
 - reentrant 267
 - SQLCODE host variable 253
 - SQLSTATE host variable 253
 - variable declaration 255
- assignment, compatibility rules 460
- ATTACH precompiler option 933
- attachment facility
 - options in z/OS environment 73
- AUTH SIGNON (connection function of RRSF)
 - language examples 128
 - syntax 128
- authority
 - authorization ID 1029
 - creating test tables 1033
 - SYSIBM.SYSTABAUTH table 677
- AUTO COMMIT field of SPUFI panel 1039
- automatic query rewrite 473
- automatic rebind
 - conditions for 970
 - invalid package 970
 - SQLCA not available 970

B

- batch processing
 - access to DB2 and DL/I together
 - binding a plan 956
 - checkpoint calls 772
 - commits 772
 - precompiling 920
 - batch DB2 application
 - running 1029
 - starting with a CLIST 1030
- bill of materials applications 489
- binary host variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - PL/I 409
- binary host variable array
 - C/C++ 287
 - PL/I 415
- binary large object (BLOB) 465
- BIND
 - command line processor command 946
- BIND COPY
 - for native SQL procedures 591
- BIND COPY REPLACE
 - for native SQL procedures 592
- bind options
 - planning for 53
- BIND PACKAGE subcommand of DSN
 - options
 - CURRENTDATA 955
 - ENCODING 956
 - location-name 955

- BIND PACKAGE subcommand of DSN (*continued*)
 - options (*continued*)
 - OPTIONS 955
 - SQLERROR 955
 - options associated with DRDA access 954, 956
 - remote 944
- BIND PLAN subcommand of DSN
 - options
 - CURRENTDATA 955
 - DISCONNECT 954
 - ENCODING 955
 - SQLRULES 954, 972
 - options associated with DRDA access 954
 - remote 944
- binding
 - application plans 943
 - changes that require 52
 - checking BIND PACKAGE options 956
 - DBRMs precompiled elsewhere 920
 - options associated with DRDA access 954
 - packages
 - remote 944
 - plans 949
 - remote package requirements 944
 - specify SQL rules 972
- block fetch
 - preventing 722
 - with cursor stability 722
- BMP (batch message processing) program
 - checkpoints 62
- bounded character pointer host variable
 - declaring 300
 - description 300
 - referencing in SQL statements 298
- BTS (batch terminal simulator) 1066

C

- C application program
 - declaring tables 307
 - sample application 1094
- C/C++
 - creating stored procedure 615
- C/C++ application program
 - data type compatibility 301
 - DCLGEN support 167
 - declaring views 307
 - defining the SQLDA 173, 274
 - host structure 295
 - INCLUDE statement 307
 - including SQLCA 273
 - indicator variable array declaration 297
 - indicator variable declaration 297
 - naming convention 307
 - precompiler option defaults 940
 - SQLCODE host variable 273
 - SQLSTATE host variable 273
 - variable array declaration 287
 - variable declaration 275
 - with classes, preparing 920
- C/C++ application programs
 - pointer host variables 300
- CAF (call attachment facility)
 - description 77
- CAF functions
 - summary of behavior 85

- calculated values
 - groups with conditions 693
 - summarizing group values 692
- call attachment facility (CAF)
 - application program
 - examples 100
 - preparation 81
 - attention exit routines 79
 - authorization IDs 78
 - behavior summary 85
 - connection functions 86
 - connection name 78
 - connection properties 78
 - connection type 78
 - DB2 abends 78
 - description 77
 - error messages 98
 - implicit connections to 82
 - invoking 74
 - parameters for CALL DSNALI 83
 - program requirements 81
 - recovery routines 79
 - register changes 82
 - return codes
 - example of checking 100
 - return codes and reason codes 98
 - sample scenarios 99
 - scope 78
 - terminated task 78
 - trace 98
- call attachment language interface
 - loading 80
 - making available 80
- CALL DSNALI
 - parameter list 83
 - required parameters 83
- CALL DSNRLI
 - parameter list 115
 - required parameters 115
- CALL statement
 - command line processor 1028
 - examples 791
 - multiple 803
 - syntax for invoking DSNTPSMP 605
- catalog table
 - SYSIBM.LOCATIONS 875
 - SYSIBM.SYSCOLUMNS 677
 - SYSIBM.SYSTABAUTH 677
- CCSID (coded character set identifier)
 - controlling in COBOL programs 351
 - precompiler option 933
 - setting for host variables 178
 - SQLDA 202
- CEEDUMP
 - using to debug stored procedures 1052
- character host variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - Fortran 397
 - PL/I 409
- character host variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
- character input data
 - REXX program 444
- character large object (CLOB) 465
- character string
 - literals 251
 - mixed data 460
 - width of column in results 1043, 1049
- check constraint
 - check integrity 471
 - considerations 469
 - CURRENT RULES special register effect 470
 - defining 469
 - description 469
 - determining violations 1070
 - enforcement 469
 - programming considerations 1070
- CHECK-pending status 471
- checkpoint
 - calls 61, 62
 - specifying frequency 62
- CHKP call, IMS 61
- CICS
 - DSNTIAC subroutine
 - assembler 267
 - C 307
 - COBOL 357
 - PL/I 427
 - environment planning 1030
 - facilities
 - command language translator 929
 - control areas 1017
 - EDF (execution diagnostic facility) 1066
 - language interface module (DSNCLI)
 - use in link-editing an application 942
 - operating
 - running a program 1017
 - preparing with JCL procedures 979
 - programming
 - DFHEIENT macro 267
 - sample applications 1096, 1100
 - SYNCPOINT command 57
 - storage handling
 - assembler 267
 - C 307
 - COBOL 357
 - PL/I 427
 - sync point 57
 - unit of work 57
- CICS applications
 - thread reuse 157
- CICS attachment facility
 - controlling from applications 155
 - detecting whether it is operational 156
 - starting 155
 - stopping 155
- client 68
- client program
 - preparing for calling a remote stored procedure 800
- CLOSE
 - statement
 - description 731
 - recommendation 736
 - WHENEVER NOT FOUND clause 200, 202
- CLOSE (connection function of CAF)
 - description 86
 - language examples 93
 - program example 100
 - syntax 93
- COALESCE function 703

- COBOL
 - creating stored procedure 615
- COBOL application program
 - compiling 942
 - controlling CCSID 351
 - data type compatibility 352
 - DB2 precompiler option defaults 940
 - DCLGEN support 167
 - declaring tables 357
 - declaring views 357
 - defining the SQLDA 173, 324
 - dynamic SQL 198
 - host structure 345
 - host variable
 - use of hyphens 357
 - host variable array, declaring 325
 - host variable, declaring 325
 - INCLUDE statement 357
 - including SQLCA 323
 - indicator variable array declaration 349
 - indicator variable declaration 349
 - naming convention 357
 - object-oriented extensions 363
 - options 357
 - preparation 942
 - resetting SQL-INIT-FLAG 357
 - sample program 364
 - SQLCODE host variable 323
 - SQLSTATE host variable 323
 - variable array declaration 335
 - variable declaration 326
 - WHENEVER statement 357
 - with classes, preparing 920
- coding SQL statements
 - dynamic 193
- collection, package
 - identifying 951
 - SET CURRENT PACKAGESET statement 951
- colon
 - preceding a host variable 183
 - preceding a host variable array 191
- column
 - data types 460
 - default value
 - system-defined 459
 - user-defined 460
 - displaying, list of 677
 - heading created by SPUFI 1049
 - labels, usage 202
 - name, with UPDATE statement 670
 - retrieving, with SELECT 678
 - specified in CREATE TABLE 459
 - width of results 1043, 1049
- COMMA precompiler option 933
- command line processor
 - binding 946
 - CALL statement 1028
 - stored procedures 1028
- commit point
 - description 56
 - IMS unit of work 61
- COMMIT statement
 - description 1039
 - in a stored procedure 571
 - when to issue 56
 - with RRSAF 108
- common table expressions
 - description 488
 - examples 489
 - in a CREATE VIEW statement 487
 - in a SELECT statement 487
 - in an INSERT statement 487
 - infinite loops 707
 - recursion 489
- comparison
 - compatibility rules 460
 - HAVING clause
 - subquery 712
 - operator, subquery 712
 - WHERE clause
 - subquery 712
- compatibility
 - data types 460
 - rules 460
- composite key 476
- compound statement
 - example
 - dynamic SQL 569
 - nested IF and WHILE statements 567
 - EXIT handler 579
 - labels 566
- compound statements
 - nested 575
 - within the declaration of a condition handler 579
- condition handlers
 - empty 587
- conditions
 - ignoring 587
- CONNECT
 - statement
 - SPUFI 1039
- CONNECT (connection function of CAF)
 - description 86
 - language examples 87
 - program example 100
 - syntax 87
- CONNECT LOCATION field of SPUFI panel 1039
- CONNECT precompiler option 933
- CONNECT processing option
 - enforcing restricted system rules 69
- CONNECT statement, with DRDA access 874
- connecting
 - DB2 73
- connection
 - DB2
 - connecting from tasks 1021
 - function of CAF
 - CLOSE 93
 - CONNECT 87
 - DISCONNECT 95
 - OPEN 91
 - TRANSLATE 96
 - function of RRSAF
 - AUTH SIGNON 128
 - CONTEXT SIGNON 132
 - CREATE THREAD 141
 - FIND_DB2_SYSTEMS 147
 - IDENTIFY 118
 - SET_CLIENT_ID 137
 - SET_ID 136
 - SET_REPLICATION 140
 - SIGNON 124
 - SWITCH TO 121

- connection (*continued*)
 - function of RRSAF (*continued*)
 - TERMINATE IDENTIFY 144
 - TERMINATE THREAD 143
 - TRANSLATE 146
- connection properties
 - call attachment facility (CAF) 78
 - Resource Recovery Services attachment facility (RRSAF) 110
- connection to DB2
 - environment requirements 73
- constants, syntax
 - C/C++ 275
 - Fortran 397
- CONTEXT SIGNON (connection function of RRSAF)
 - language examples 132
 - syntax 132
- CONTINUE clause of WHENEVER statement 233
- CONTINUE handler (SQL procedure)
 - description 579
 - example 579
- coordinating updates
 - distributed data 69
- correlated reference
 - correlation name 716
 - SQL rules 695
 - usage 695
 - using in subquery 716
- correlation name 716
- create
 - external SQL procedure by using DSNTPSMP 601
 - external SQL procedure by using JCL 612
 - external stored procedure 615
- CREATE GLOBAL TEMPORARY TABLE statement 479
- CREATE PROCEDURE statement
 - external stored procedure 615
 - for external SQL procedures 612
- CREATE TABLE statement
 - DEFAULT clause 459
 - NOT NULL clause 459
 - PRIMARY KEY clause 474
 - relationship names 476
 - UNIQUE clause 459, 474
 - usage 459
- CREATE THREAD (connection function of RRSAF)
 - language examples 141
 - program example 151
 - syntax 141
- CREATE TRIGGER
 - activation order 506
 - description 493
 - example 493
 - timestamp 506
 - trigger naming 493
- CREATE TYPE statement
 - example 513
- CREATE VIEW statement 485
- created temporary table
 - instances 480
 - use of NOT NULL 480
 - working with 481
- creating objects
 - in an application program 459
- creating stored procedures
 - external SQL procedures 600
- CURRENT PACKAGESET special register
 - dynamic plan switching 961
- CURRENT PACKAGESET special register (*continued*)
 - identify package collection 951
- CURRENT RULES special register
 - effect on check constraints 470
 - usage 972
- current server 68
- CURRENT SERVER special register
 - description 951
 - saving value in application program 876
- CURRENT SQLID special register
 - use in test 1031
 - value in INSERT statement 460
- cursor
 - attributes
 - using GET DIAGNOSTICS 746
 - using SQLCA 745
 - closing 731
 - CLOSE statement 736
 - deleting a current row 733
 - description 722
 - dynamic scrollable 723
 - effect of abend on position 725
 - example
 - retrieving backward with scrollable cursor 749
 - updating specific row with rowset-positioned cursor 752
 - updating with non-scrollable cursor 749
 - updating with rowset-positioned cursor 751
 - insensitive scrollable 723
 - maintaining position 725
 - non-scrollable 723
 - open state 725
 - OPEN statement 729
 - result table 722
 - row-positioned
 - declaring 727
 - deleting a current row 729
 - description 722
 - end-of-data condition 729
 - retrieving a row of data 729
 - steps in using 726
 - updating a current row 729
 - rowset-positioned
 - declaring 732
 - description 722
 - end-of-data condition 732
 - number of rows 733
 - number of rows in rowset 736
 - opening 732
 - retrieving a rowset of data 733
 - steps in using 731
 - updating a current rowset 733
 - scrollable
 - description 723
 - dynamic 723
 - fetch orientation 737
 - INSENSITIVE 723
 - retrieving rows 737
 - SENSITIVE DYNAMIC 723
 - SENSITIVE STATIC 723
 - sensitivity 723
 - static 723
 - updatable 723
 - static scrollable 723
 - types 723
- WITH HOLD
 - description 725

cursors
 declaring in SQL procedures 577

D

data

 accessing from an application program 677
 adding 655
 adding to the end of a table 670
 associated with WHERE clause 678
 currency 722
 distributed 68
 modifying 655
 not in a table 771
 retrieval using SELECT * 709
 retrieving a rowset 733
 retrieving a set of rows 729
 retrieving large volumes 770
 scrolling backward through 746
 security and integrity 55
 updating during retrieval 708
 updating previously retrieved data 748

data encryption 477

data integrity

 tables 468

data type

 built-in 460
 comparisons 183
 compatibility
 assembler application program 261
 C application program 301
 COBOL and SQL 352
 Fortran and SQL 401
 PL/I application program 423
 REXX and SQL 438

data types

 compatibility 180
 used by DCLGEN 167

DATE precompiler option 933

datetime data type 460

DB2

 connection from a program 73

DB2abend

 DL/I batch 772

DB2 coprocessor 926

 processing SQL statements 918

DB2 environment information block, retrieving

 DSNTWRE 819

DB2 MQ tables

 descriptions 884

DB2 private protocol access

 coding an application 871

DB2_RETURN_STATUS

 using to get procedure status 806

DB2-supplied stored procedures 811

DB2I

 default panels 917
 invoking DCLGEN 162

DB2I (DB2 Interactive)

 background processing
 run time libraries 987
 EDITJCL processing
 run time libraries 987
 interrupting 1035
 menu 1035
 panels
 BIND PACKAGE 993

DB2I (DB2 Interactive) (continued)

panels (continued)

 BIND PLAN 996
 Compile, Link, and Run 1005
 Current SPUFI Defaults 1041
 DB2I Primary Option Menu 981, 1035
 Defaults for BIND PACKAGE 999
 Defaults for BIND PLAN 1001
 Defaults for REBIND PACKAGE 999
 Defaults for REBIND PLAN 1001
 Precompile 990
 Program Preparation 983
 System Connection Types 1003
 preparing programs 915
 program preparation example 983
 selecting
 SPUFI 1035
 SPUFI 1035

DB2I defaults

 setting 917

DBCS (double-byte character set)

 translation in CICS 929

DBINFO

 passing to external stored procedure 615
 user-defined function 537

DBRM (database request module) 949

 binding to a package 944
 description 925

DBRMs in HFS files

 binding 946

DCLGEN

 COBOL example 170
 data types 167
 declaring indicator variable arrays 162
 generating table and view declarations 161
 generating table and view declarations from DB2I 162
 INCLUDE statement 169
 including declarations in a program 169
 invoking 161
 using from DB2I 162
 variable declarations 167

DCLGEN (declarations generator)

 description 161

DDITV02 input data set 975

DDOTV02 output data set 975

Debug Tool

 user-defined function 1050

debugging

 recording messages for stored procedures 1058
 stored procedures 1052

debugging application programs 1059

DEC15

 precompiler option 933
 rules 709

DEC31

 avoiding overflow 709
 precompiler option 933
 rules 709

decimal

 15 digit precision 709
 31 digit precision 709
 arithmetic 709

DECIMAL data type

 C/C++ 275

declarations generator (DCLGEN)

 description 161

- DECLARE CURSOR statement
 - description, row-positioned 727
 - description, rowset-positioned 732
 - FOR UPDATE clause 727
 - multilevel security 727
 - prepared statement 200, 202
 - scrollable cursor 723
 - WITH HOLD clause 725
 - WITH RETURN option 641
 - WITH ROWSET POSITIONING clause 732
- DECLARE GLOBAL TEMPORARY TABLE statement 481
- DECLARE TABLE statement
 - assembler 267
 - C 307
 - COBOL 357
 - Fortran 403
 - in application programs 160
 - PL/I 427
- declared temporary table
 - including column defaults 481
 - including identity columns 482
 - instances 481
 - ON COMMIT clause 483
 - qualifier for 481
 - remote access using a three-part name 871
 - requirements 481
 - working with 481
- declaring tables and views
 - advantages 160
- DELETE statement
 - correlated subquery 716
 - description 661, 673
 - positioned
 - FOR ROW n OF ROWSET clause 733
 - restrictions 729
 - WHERE CURRENT clause 729, 733
- deleting
 - current rows 729
 - data 673
 - every row from a table 673
 - with TRUNCATE 673
 - rows from a table 673
- delimiter, SQL 183
- DENSE_RANK specification 686
 - example 686
- department sample table 1072
 - creating 478
- DEPLOY bind option
 - for native SQL procedures 595
- DESCRIBE INPUT statement 226
- DESCRIBE statement
 - column labels 202
 - INTO clauses 202
- designing
 - applications 1
- designing applications
 - distributed data 67
- DFHEIENT macro 267
- DFSLI000 (IMS language interface module) 942
- diagnostics area
 - RESIGNAL affect on 591
 - SIGNAL affect on 591
- direct row access 754
- disability xvi
- DISCONNECT (connection function of CAF)
 - description 86
 - language examples 95
- DISCONNECT (connection function of CAF) (continued)
 - program example 100
 - syntax 95
- displaying
 - table columns 677
 - table privileges 677
- DISTINCT
 - clause of SELECT statement 682
 - unique values 682
- distinct type
 - assigning values 659
 - comparing types 719
 - description 513
 - example
 - argument of user-defined function (UDF) 514
 - arguments of infix operator 788
 - casting constants 788
 - casting function arguments 788
 - casting host variables 788
 - LOB data type 514
 - function arguments 787
 - strong typing 513
 - UNION with INTERSECT 719
 - with EXCEPT 719
 - with UNION 719
- distinct types
 - creating 513
- distributed data 68
 - coordinating updates 69
 - copying a remote table 871
 - DBPROTOCOL bind option 871, 953
 - designing applications for 67
 - encoding scheme of retrieved data 877
 - example
 - accessing remote temporary table 873
 - calling stored procedure at remote location 953
 - connecting to remote server 874, 953
 - specifying location in table name 953
 - using alias for multiple sites 875
 - using RELEASE statement 876
 - using three-part table names 871
 - executing long SQL statements 877
 - identifying server at run time 876
 - maintaining data currency 722
 - planning
 - DB2 private protocol access 953
 - DRDA access 953
 - program preparation 956
 - programming
 - coding with DB2 private protocol access 871
 - coding with DRDA access 871
 - retrieving from ASCII or Unicode tables 877
 - three-part table names 871
 - transmitting mixed data 876
 - two-phase commit 69
 - using alias for location 875
- DL/I batch
 - application programming 772
 - checkpoint ID 1027
 - DB2 requirements 772
 - DDITV02 input data set 975
 - DSNMTV01 module 1024
 - features 772
 - SSM= parameter 1024
 - submitting an application 1024
- double-byte character large object (DBCLOB) 465

- DRDA access
 - accessing remote temporary table 873
 - bind options 954
 - coding an application 871
 - connecting to remote server 874
 - planning 953
 - precompiler options 942
 - preparing programs 954
 - programming hints 876
 - releasing connections 876
 - sample program 375
 - SQL limitations at different servers 876
- DRDA access with CONNECT statements
 - sample program 375
- DRDA with three-part names
 - sample program 381
- DROP TABLE statement 484
- DSN applications, running with CAF 74
- DSN command of TSO
 - return code processing 1017
 - RUN subcommands 1017
- DSN_FUNCTION_TABLE table 783
- DSN_WLM_APPLENV procedure 822
- DSN8BC3 sample program 357
- DSN8BD3 sample program 307
- DSN8BE3 sample program 307
- DSN8BF3 sample program 403
- DSN8BP3 sample program 427
- DSNACCOR stored procedure
 - description 842
 - option descriptions 844
 - output 859
 - syntax diagram 843
- DSNACICS stored procedure 825
- DSNACICX exit routine 831
- DSNAIMS stored procedure 833
- DSNAIMS2 stored procedure 838
- DSNALI
 - loading 80
 - making available 80
- DSNALI (CAF language interface module)
 - example of deleting 100
 - example of loading 100
- DSNCLI (CICS language interface module) 942
- DSNEBP10 999
- DSNEBP11 999
- DSNH command of TSO 1061
- DSNHASM procedure 978
- DSNHC procedure 978
- DSNHCOB procedure 978
- DSNHCOB2 procedure 978
- DSNHCPP procedure 978
- DSNHCPP2 procedure 978
- DSNHFOR procedure 978
- DSNHICB2 procedure 978
- DSNHICOB procedure 978
- DSNHILI entry point to DSNALI
 - program example 100
- DSNHILI2 entry point to DSNALI
 - program example 100
- DSNHPLI procedure 978
- DSNMTV01 module 1024
- DSNRLI
 - loading 112
 - making available 112
- DSNTEEDIT CLIST 965
- DSNTEP2 and DSNTEP4 sample program
 - specifying SQL terminator 1101, 1108
- DSNTEP2 sample program
 - how to run 1092
 - parameters 1092
 - program preparation 1092
- DSNTEP4 sample program
 - how to run 1092
 - parameters 1092
 - program preparation 1092
- DSNTIAC subroutine
 - assembler 267
 - C 307
 - COBOL 357
 - PL/I 427
- DSNTIAD sample program
 - how to run 1092
 - parameters 1092
 - program preparation 1092
 - specifying SQL terminator 1106
- DSNTIAR subroutine
 - assembler 239
 - C 307
 - COBOL 357
 - description 229
 - Fortran 403
 - PL/I 427
 - return codes 231
 - using 229
- DSNTIAUL sample program
 - how to run 1092
 - parameters 1092
 - program preparation 1092
- DSNTIJS sample program
 - using to set up the Unified Debugger 1056
- DSNTIR subroutine 403
- DSNTPSMP
 - creating external SQL procedures 601
 - required authorizations 601
 - syntax for invoking 605
- DSNTRACE data set 98
- DSNTWRE
 - description 819
- DSNULI 155
- DSNXDBRM 925
- DSNXNBRM 925
- DYNAM option of COBOL 357
- dynamic buffer allocation
 - FETCH WITH CONTINUE 743
- dynamic plan selection
 - restrictions with CURRENT PACKAGESET special register 961
 - using packages with 961
- dynamic SQL
 - advantages and disadvantages 194
 - assembler program 202
 - C program 202
 - COBOL application program 357
 - COBOL program 198
 - description 193
 - effect of bind option REOPT(ALWAYS) 202
 - effect of WITH HOLD cursor 221
 - EXECUTE IMMEDIATE statement 219
 - fixed-list SELECT statements 200
 - Fortran program 403
 - host languages 198
 - non-SELECT statements 199, 221

- dynamic SQL (*continued*)
 - PL/I 202
 - PREPARE and EXECUTE 221
 - programming 193
 - requirements 194
 - restrictions 194
 - sample C program 311
 - varying-list SELECT statements 202
- DYNAMICRULES bind option 959

E

- ECB (event control block)
 - CONNECT function of CAF 87
 - IDENTIFY function of RRSAP 118
 - SET_REPLICATION function of RRSAP 140
- EDIT panel, SPUFI
 - SQL statements 1035
- embedded semicolon
 - embedded 1106
- embedded SQL applications
 - host variables, XML data 241
 - XML data 241
- employee photo and resume sample table 1077
- employee sample table 1074
- employee-to-project activity sample table 1081
- ENCRYPT_TDES function 477
- END-EXEC delimiter 183
- end-of-data condition 729, 732
- error
 - arithmetic expression 240
 - division by zero 240
 - handling 233
 - messages generated by precompiler 1060, 1061
 - overflow 240
 - return codes 228
 - run 1059
- errors when retrieving data into a host variable
 - determining cause 179
- EXCEPT
 - eliminating duplicate rows 689
 - keeping duplicate rows with ALL 691
- EXCEPT clause
 - columns of result table 688
- exception condition handling 233
- EXEC SQL delimiter 183
- EXECUTE IMMEDIATE statement 219
- EXECUTE statement
 - dynamic execution 221
 - parameter types 202
 - USING DESCRIPTOR clause 202
- EXISTS predicate, subquery 712
- EXIT handler (SQL procedure) 579
- exit routine
 - abend recovery with CAF 79
 - attention processing with CAF 79
- exit routines
 - DSNACICX 831
- EXPLAIN
 - automatic rebind 970
- EXPLAIN tables
 - DSN_FUNCTION_TABLE 784
- external SQL procedure
 - creating 600
- external SQL procedures 565
 - creating by using DSNTPSMP 601
 - creating by using JCL 612

- external SQL procedures (*continued*)
 - debugging with the Unified Debugger 1056
 - migrating to native SQL procedures 596
- external stored procedure
 - creating 615
 - modifying the definition 653
 - package 638
 - package authorizations 638
 - plan 638
 - preparing 615
 - reentrant 643
 - running as authorized program 615

F

- FETCH CURRENT CONTINUE 742
- FETCH statement
 - description, multiple rows 733
 - description, single row 729
 - fetch orientation 737
 - host variables 200
 - multiple-row
 - assembler 267
 - description 733
 - FOR n ROWS clause 736
 - number of rows in rowset 736
 - using with descriptor 733
 - using with host variable arrays 733
 - row and rowset positioning 748
 - scrolling through data 746
 - USING DESCRIPTOR clause 202
 - using row-positioned cursor 729
- FETCH WITH CONTINUE 742
- file reference variable 767
 - DB2-generated construct 767
- FIND_DB2_SYSTEMS (connection function of RRSAP)
 - language examples 147
 - syntax 147
- fixed buffer allocation
 - FETCH WITH CONTINUE 744
- FLAG precompiler option 933
- FLOAT precompiler option 933
- FOLD
 - value for C and CPP 933
 - value of precompiler option HOST 933
- FOR UPDATE clause 727
- FOREIGN KEY clause
 - description 476
 - usage 476
- format
 - SELECT statement results 1048
 - SQL in input data set 1035
- formatting
 - result tables 682
- Fortran application program
 - @PROCESS statement 403
 - byte data type 403
 - constant syntax 397
 - data type compatibility 401
 - declaring tables 403
 - declaring views 403
 - defining the SQLDA 173, 396
 - host variable, declaring 397
 - INCLUDE statement 403
 - including SQLCA 395
 - indicator variable declaration 400
 - naming convention 403

- Fortran application program (*continued*)
 - parallel option 403
 - precompiler option defaults 940
 - SQLCODE host variable 395
 - SQLSTATE host variable 395
 - statement labels 403
 - variable declaration 397
 - WHENEVER statement 403
- FROM clause
 - joining tables 695
 - SELECT statement 678
- FRR (functional recovery routine)
 - in CAF 79
- FULL OUTER JOIN clause 702
- function resolution 780
- functional recovery routine (FRR)
 - in CAF 79

G

- GENERAL linkage convention 619, 622
- GENERAL WITH NULLS linkage convention 619, 625
- general-use programming information, described 1119
- generating table and view declarations
 - by using DCLGEN 161
 - with DCLGEN from DB2I 162
- generating XML documents for MQ message queue 883
- GET DIAGNOSTICS
 - using to get procedure status 806
- GET DIAGNOSTICS statement
 - condition items 234
 - connection items 234
 - data types for items 234, 236
 - description 234
 - multiple-row INSERT 234
 - RETURN_STATUS item 588
 - ROW_COUNT item 733
 - statement items 234
 - using in handler 587
- GO TO clause of WHENEVER statement 233
- GRANT statement 1033
- graphic host variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - PL/I 409
- graphic host variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
- GRAPHIC precompiler option 933
- GROUP BY clause
 - use with aggregate functions 692

H

- handler, using in SQL procedure 578
- HAVING clause
 - selecting groups subject to conditions 693
- HOST
 - FOLD value for C and CPP 933
 - precompiler option 933
- host language
 - dynamic SQL 198
- host language data types
 - compatibility with SQL data types 180

- host structure
 - C/C++ 295
 - COBOL 345
 - description 175
 - indicator structure 176
 - PL/I 420
 - retrieving row of data 193
 - using SELECT INTO 193
- host variable
 - assembler 254, 255
 - C/C++ 275
 - COBOL 325, 326
 - description 174
 - FETCH statement 200
 - Fortran 397
 - indicator variable 176
 - inserting values into tables 190
 - LOB
 - assembler 758
 - C 759
 - COBOL 760
 - Fortran 761
 - PL/I 761
 - PL/I 408, 409
 - PREPARE statement 200
 - retrieving a single row 184
 - setting the CCSID 178
 - static SQL flexibility 194
 - updating values in tables 189
 - using 183
- host variable array
 - C/C++ 287
 - COBOL 325, 335
 - description 175, 191
 - indicator variable array 176
 - inserting multiple rows 192
 - PL/I 408, 415
 - retrieving multiple rows 192
- host variable processing
 - errors 179
- host variables 174
 - compatible data types 180
 - XML in assembler 242
 - XML in C language 243
 - XML in COBOL 244
 - XML in embedded SQL applications 241
 - XML in PL/I 245

I

- IBM Data Studio Developer
 - creating external SQL procedures 600
 - creating native SQL procedures 572
- IDENTIFY (connection function of RRSAP)
 - language examples 118
 - program example 151
 - syntax 118
- identity column
 - defining 466, 659
 - IDENTITY_VAL_LOCAL function 466
 - inserting in table 483
 - inserting values into 659
 - trigger 493
 - using as parent key 466
- IKJEFT01 terminal monitor program in TSO 1029
- implicit CAF connection 82
- implicit RRSAP connections 115

- IMS
 - checkpoint calls 61
 - checkpoints 62
 - CHKP call 61
 - commit point 61
 - environment planning 1030
 - language interface module (DFSLI000) 942
 - link-editing 942
 - recovery 58
 - ROLB call 58, 61
 - ROLL call 58, 61
 - SYNC call 61
 - unit of work 61
- IMS programs
 - recovery 64
- IN predicate, subquery 712
- incompatibilities of releases
 - Version 8 migration
 - applications and SQL 1
 - Version 9.1 migration
 - applications and SQL 32
- index
 - types
 - foreign key 476
 - primary 484
 - unique 484
 - unique on primary key 475
- indicator structure
 - description 176
- indicator variable
 - description 176
 - inserting null values 190
- indicator variable array
 - description 176
 - inserting null values 190
- indicator variable arrays
 - declaring with DCLGEN 162
- indicator variable arraysC/C++ syntax 297
- indicator variable arraysCOBOL syntax 349
- indicator variable arraysPL/I syntax 422
- indicator variables
 - using to pass large output parameters 796
- indicator variablesassembler syntax 260
- indicator variablesC/C++ syntax 297
- indicator variablesCOBOL syntax 349
- indicator variablesFortran syntax 400
- indicator variablesPL/I syntax 422
- infinite loop 707
- informational referential constraint
 - automatic query rewrite 473
 - description 473
- INNER JOIN clause 699
- input data set DDITV02 975
- input parameters
 - stored procedures 561
- INSERT statement
 - description 655
 - multiple rows 657
 - single row 655
 - VALUES clause 655
 - with identity column 659
 - with ROWID column 658
- inserting
 - values from host variable arrays 192
- inserting data
 - by using host variables 190
- Interactive System Productivity Facility (ISPF) 1035

- internal resource lock manager (IRLM) 1024
- INTERSECT
 - eliminating duplicate rows 689
 - keeping duplicate rows with ALL 691
- INTERSECT clause
 - columns of result table 688
- invalid SQL terminator characters 1106
- invoking
 - call attachment facility (CAF) 74
 - Resource Recovery Services attachment facility (RRSAF) 106
- invoking stored procedures
 - syntax for command line processor 1028
- isolation level
 - REXX 445
- ISPF (Interactive System Productivity Facility)
 - browse 1039, 1048
 - DB2 uses dialog management 1035
 - DB2I Primary Option Menu 981
 - Program Preparation panel 983
 - programming 1020, 1021
 - scroll command 1049
- ISPLINK SELECT services 1020

J

- Java stored procedures
 - debugging with the Unified Debugger 1056
- JCL (job control language)
 - batch backout example 1025
 - DDNAME list format 924
 - page number format 925
 - precompilation procedures 978
 - precompiler option list format 924
 - preparing a CICS program 979
 - preparing a object-oriented program 920
 - starting a TSO batch application 1029
- join operation
 - FULL OUTER JOIN 702
 - INNER JOIN 699
 - joining a table to itself 699
 - joining tables 695
 - LEFT OUTER JOIN 703
 - more than one join 697
 - more than one join type 698
 - operand
 - nested table expression 695
 - user-defined table function 695
 - RIGHT OUTER JOIN 704
 - SQL rules 705

K

- key
 - composite 476
 - foreign 476
 - parent 475
 - primary
 - choosing 475
 - defining 474
 - recommendations for defining 474
 - using timestamp 475
 - unique 483

L

- label, column 202
- language interface modules
 - DSNCLI 942
- large object (LOB)
 - character conversion 765
 - declaring host variables 757
 - for precompiler 757
 - declaring LOB file reference variables 757
 - declaring LOB locators 757
 - defining and moving data into DB2 463
 - description 465
 - expression 765
 - file reference variable 767
 - indicator variable 764
 - locator 763
 - materialization 762
 - sample applications 756
- LEFT OUTER JOIN clause 703
- LEVEL precompiler option 933
- libraries
 - for table declarations and host-variable structures 170
- LINECOUNT precompiler option 933
- link-edit 155
- link-editing 942
 - AMODE option 1005
 - RMODE option 1005
- linkage conventions
 - GENERAL 619, 622
 - GENERAL WITH NULLS 619, 625
 - SQL 619, 629
 - stored procedures 619
- LOAD z/OS macro used by CAF 81
- LOAD z/OS macro used by RRSAF 114
- LOB column, definition 463
- LOB file reference variable
 - assembler 255
 - C/C++ 275, 287
 - COBOL 326, 335
 - PL/I 409, 415
- LOB host variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
- LOB locator
 - assembler 255
 - C/C++ 275, 287
 - COBOL 335
 - Fortran 397
 - PL/I 409, 415
- LOB values
 - fetching 742
- LOB variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - Fortran 397
 - PL/I 409
- location name 68
- lock
 - escalation
 - when retrieving large numbers of rows 770

M

- mapping macro
 - assembler applications 271
 - DSNXDBRM 925
 - DSNXNBRM 925
- MARGINS precompiler option 933
- Mashup Center
 - creating a feed 70
- materialization
 - LOBs 762
- merging
 - data 661
- message
 - analyzing 1060
 - obtaining text
 - assembler 239
 - C 307
 - COBOL 357
 - Fortran 403
 - PL/I 427
- message data
 - WebSphere MQ 879
- message handling
 - WebSphere MQ 879
- Message Queue Interface (MQI)
 - DB2 MQ tables 884
 - policies 881
 - services 880
 - WebSphere MQ 880
- messages
 - WebSphere MQ 879
- migrating
 - applications 1
- mixed data
 - converting 876
 - description 460
 - transmitting to remote location 876
- MLS (multilevel security)
 - referential constraints 473
 - triggers 508
- modified source statements 925
- modify
 - external stored procedure definition 653
- modifying
 - data 655
- MPP program
 - checkpoints 62
- MQ message queue
 - sending table data 883
 - shredding XML documents 884
- MQ XML composition stored procedures
 - alternative method 883
- MQ XML decomposition stored procedures
 - alternative method 884
- MQSeries
 - DB2 functions
 - connecting applications 895
 - MQREAD 881
 - MQREADALL 881
 - MQREADALLCLOB 881
 - MQREADCLOB 881
 - MQRECEIVE 881
 - MQRECEIVEALL 881
 - MQRECEIVEALLCLOB 881
 - MQRECEIVECLOB 881
 - MQSEND 881
 - programming considerations 881

- MQSeries (*continued*)
 - DB2 functions (*continued*)
 - retrieving messages 894
 - sending messages 893
 - DB2 scalar functions 881
 - DB2 table functions 881
- MSGFILE run time option
 - using to debug stored procedures 1058
- multilevel security (MLS) check
 - referential constraints 473
 - triggers 508
- multiple-row FETCH statement
 - checking DB2_LAST_ROW 236
 - SQLCODE +100 228
- multiple-row INSERT statement
 - dynamic execution 223
 - NOT ATOMIC CONTINUE ON SQLEXCEPTION 234
 - using GET DIAGNOSTICS 234

N

- naming convention
 - assembler 267
 - C 307
 - COBOL 357
 - Fortran 403
 - PL/I 427
 - REXX 439
 - tables you create 478
- native SQL procedures 565
 - BIND COPY 591
 - BIND COPY REPLACE 592
 - creating 572
 - debugging with the Unified Debugger 1056
 - deploying to another server 595
 - deploying to production 595
 - migrating from external SQL procedures 596
 - packages for 591
 - replacing packages for 592
- nested compound statements
 - cursor declarations 577
 - definition 575
 - for controlling scope of conditions 580
 - scope of variables 574
 - statement labels 576
- nested table expression
 - correlated reference 695
 - correlation name 695
 - join operation 695
- NEWFUN
 - precompiler option 933
- NODYNAM option of COBOL 357
- NOFOR precompiler option 933
- NOGRAPHIC precompiler option 933
- non-DB2 resources
 - accessing from stored procedure 639
- nontabular data storage 670
- NOOPTIONS precompiler option 933
- NOPADNTSTR precompiler option 933
- NOSOURCE precompiler option 933
- NOT FOUND clause of WHENEVER statement 233
- not logged
 - table spaces
 - recovering 66
- NOXREF precompiler option 933
- NUL character in C 307

- null
 - determining value of output host variable 186
- NULL
 - pointer in C 307
- null value
 - column value of UPDATE statement 670
 - determining column value 188
 - inserting into columns 190
- Null, in REXX 439
- numeric
 - data
 - width of column in results 1049
- numeric data
 - description 460
 - width of column in results 1043
- numeric host variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - Fortran 397
 - PL/I 409
- numeric host variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
- NUMTCB parameter 805

O

- object-oriented program, preparation 920
- objects
 - creating in a application program 459
- ON clause, joining tables 695
- ONEPASS precompiler option 933
- OPEN
 - statement
 - opening a cursor 729
 - opening a rowset cursor 732
 - prepared SELECT 200
 - USING DESCRIPTOR clause 202
 - without parameter markers 202
- OPEN (connection function of CAF)
 - description 86
 - language examples 91
 - program example 100
 - syntax 91
 - syntax usage 91
- OPTIONS precompiler option 933
- ORDER BY clause
 - SELECT statement 684
 - with ORDER OF clause 669
- ORDER OF clause 669
- organization application
 - examples 1094
- outer join
 - FULL OUTER JOIN 702
 - LEFT OUTER JOIN 703
 - RIGHT OUTER JOIN 704
- output host variable
 - determining if null 186
 - determining if truncated 186
- output host variable processing
 - errors 179
- output parameters
 - stored procedures 561, 796

P

- package
 - binding
 - DBRM to a package 943
 - remote 944
 - to plans 949
 - identifying at run time 951
 - invalid 52
 - invalidated 970
 - listing 949
 - location 951
 - rebinding examples 963
 - rebinding with pattern-matching characters 963
 - selecting 951
 - trigger 504
 - version, identifying 945
- package authorization
 - for external stored procedures 638
- packages
 - collection ID for stored procedure packages 643
 - for external procedures 638
 - for native SQL procedures 591
 - for nested routines 643
- packages bound on DB2 Version 6 and before 1, 32
- PADNTSTR precompiler option 933
- panel
 - Current SPUFI Defaults 1041, 1046
 - DB2I Primary Option Menu 1035
 - DSNEPRI 1035
 - DSNESP01 1035
 - DSNESP02 1041
 - DSNESP07 1046
 - EDIT (for SPUFI input data set) 1035
 - SPUFI 1035
- panels
 - DB2I (DB2 Interactive) 170
 - DB2I DEFAULTS 170
 - DCLGEN 170
 - DSNEDP01 170
 - DSNEOP01 170
 - DSNEOP02 170
 - REBIND PACKAGE 1009
 - REBIND TRIGGER PACKAGE 1011
- parameter list
 - stored procedures 561
- parameter marker
 - casting in function invocation 788
 - dynamic SQL 221
 - more than one 221
 - values provided by OPEN 200
 - with arbitrary statements 202
- parameter marker information
 - obtaining by using an SQLDA 226
- PARAMETER STYLE SQL option
 - using to debug stored procedures 1052
- parent key 475
- PARMS option 1020
- performance
 - affected by
 - application structure 1020
 - application programs 772
 - programming applications 54
- PERIOD precompiler option 933
- phone application, description 1094
- PL/I
 - creating stored procedure 615
 - PL/I application program
 - coding considerations 427
 - data type compatibility 423
 - DBCS constants 427
 - DCLGEN support 167
 - declaring tables 427
 - declaring views 427
 - defining the SQLDA 173, 408
 - host structure 420
 - host variable array, declaring 408
 - host variable, declaring 408
 - INCLUDE statement 427
 - including SQLCA 407
 - indicator variable array declaration 422
 - indicator variable declaration 422
 - naming convention 427
 - SQLCODE host variable 407
 - SQLSTATE host variable 407
 - statement labels 427
 - variable array declaration 415
 - variable declaration 409
 - WHENEVER statement 427
 - planning
 - applications 1
 - bind options 53
 - planning application programs
 - SQL processing options 51
 - planning applications
 - recovery 55
 - plans bound on DB2 Version 6 and before 1, 32
 - pointer host variables
 - declaring 300
 - referencing in SQL statements 298
 - policies
 - Message Queue Interface (MQI) 881
 - WebSphere MQ 879
 - precompiler
 - binding on another system 920
 - data sets used by 921
 - description 918
 - diagnostics 925
 - functions 920
 - input 923
 - maximum size of input 923
 - modified source statements 925
 - option descriptions 932
 - options
 - CONNECT 942
 - defaults 940
 - DRDA access 942
 - SQL 942
 - output 925
 - precompiling programs 918
 - running 920
 - starting
 - dynamically 924
 - JCL for procedures 978
 - submitting jobs
 - ISPF panels 983
 - submitting jobs with ISPF panels 915
 - predicate
 - general rules 678
 - PRELINK utility 983
 - PREPARE statement
 - dynamic execution 221
 - host variable 200
 - INTO clause 202

- preparing programs
 - overview 973
- PRIMARY KEY clause
 - ALTER TABLE statement 484
 - CREATE TABLE statement 474
- problem determination, guidelines 1059
- procedure status
 - retrieving 806
 - setting 806
- procedures
 - creating versions 593
 - DSN_WLM_APPLENV 822
 - inheriting special registers 548
 - WLM_SET_CLIENT_INFO 820
- product-sensitive programming information, described 1119
- production environment
 - deploying native SQL procedures 595
- program preparation 915
- program problems checklist
 - documenting error situations 1059
 - error messages 1064, 1065
- programming applications
 - performance 54
- programming interface information, described 1119
- project activity sample table 1080
- project application, description 1094
- project sample table 1079
- PSPI symbols 1119

Q

- queries
 - in application programs 159
 - tuning in application programs 772
- QUOTE precompiler option 933
- QUOTESQL precompiler option 933

R

- RANK specification 686
 - example 686
- real-time statistics
 - stored procedure 842
- reason code
 - CAF 98
 - RRSAF 148
 - X"00D44057" 772
- REBIND PACKAGE subcommand of DSN
 - generating list of 965
 - rebinding with wildcard characters 963
 - remote 944
- REBIND PLAN subcommand of DSN
 - generating list of 965
 - options
 - NOPKLIST 964
 - PKLIST 964
 - remote 944
- REBIND TRIGGER PACKAGE subcommand of DSN 504
- rebinding
 - automatically
 - conditions for 970
 - changes that require 52
 - list of plans and packages 965
 - lists of plans or packages 965
 - packages with pattern-matching characters 963
 - planning for 970

- rebinding (*continued*)
 - plans 964
- recovering
 - table spaces that are not logged 66
- recovery
 - IMS programs 58, 64
 - planning for in your application 55
- recursive SQL
 - controlling depth 489
 - description 707
 - examples 489
 - infinite loops 707
 - rules 707
 - single level explosion 489
 - summarized explosion 489
- reentrant code
 - in stored procedures 643
- referential constraint
 - defining 471
 - description 471
 - determining violations 1070
 - informational 473
 - name 476
 - on tables with data encryption 477
 - on tables with multilevel security 473
- referential integrity
 - effect on subqueries 718
 - programming considerations 1070
- register conventions
 - RRSAF 114
- registering XML schema
 - XSR_REGISTER 863
- registers
 - changed by CAF (call attachment facility) 82
- release incompatibilities
 - Version 8 migration
 - applications and SQL 1
 - Version 9.1 migration
 - applications and SQL 32
- RELEASE SAVEPOINT statement 65
- RELEASE statement, with DRDA access 876
- remote stored procedure
 - preparing client program 800
- REPLACE statement (COBOL) 357
- requester 68
- resetting control blocks
 - CAF 95
- RESIGNAL statement
 - raising a condition 588
 - setting SQLSTATE value 589
- Resource Recovery Services attachment facility (RRSAF)
 - application program
 - preparation 114
 - authorization IDs 110
 - behavior summary 116
 - connection functions 118
 - connection name 110
 - connection properties 110
 - connection type 110
 - DB2 abends 110
 - description 108
 - implicit connections 115
 - invoking 106
 - loading 112
 - making available 112
 - parameters for CALL DSNRLI 115
 - program examples 151

- Resource Recovery Services attachment facility (RRSAF)
 - (continued)
 - program requirements 114
 - register conventions 114
 - return codes and reason codes 148
 - sample JCL 151
 - sample scenarios 149
 - scope 110
 - terminated task 110
 - restart, DL/I batch programs using JCL 1025
 - restricted system
 - definition 69
 - forcing rules 69
 - update rules 69
 - restricted systems 67
 - result column
 - join operation 702
 - naming with AS clause 683
 - result set locator
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - Fortran 397
 - PL/I 409
 - result sets
 - receiving from a stored procedure 807
 - result table
 - description 682
 - example 682
 - numbering rows 685
 - of SELECT statement 682
 - read-only 727
 - result tables
 - formatting 682
 - retrieving
 - data in ASCII from DB2 for z/OS 202
 - data in Unicode from DB2 for z/OS 202
 - data using SELECT * 709
 - data, changing the CCSID 202
 - large volumes of data 770
 - multiple rows into host variable arrays 192
 - retrieving a single row
 - into host variables 184
 - retrieving DB2 environment information block
 - DSNTWRE 819
 - return code
 - CAF 98
 - DSN command 1017
 - RRSAF 148
 - RETURN statement
 - returning SQL procedure status 806
 - REXX
 - creating stored procedure 615
 - REXX application program
 - including SQLCA 437
 - SQLCODE host variable 437
 - SQLSTATE host variable 437
 - REXX program
 - application programming interface
 - CONNECT 442
 - DISCONNECT 442
 - EXECSQL 442
 - character input data 444
 - data type conversion 438
 - DSNREXX 442
 - error handling 439
 - input data type 438
 - REXX program (continued)
 - isolation level 445
 - naming convention 439
 - naming cursors 447
 - naming prepared statements 447
 - running 1020
 - SQLDA 173, 437
 - statement label 439
 - RIB (release information block)
 - CONNECT function of CAF 87
 - IDENTIFY function of RRSAF 118
 - SET_REPLICATION function of RRSAF 140
 - RID
 - for direct row access 754
 - RID function 754
 - RIGHT OUTER JOIN clause 704
 - RMODE link-edit option 1005
 - ROLB call, IMS 58, 61
 - ROLL call, IMS 58, 61
 - rollback
 - changes within a unit of work 65
 - ROLLBACK option
 - CICS SYNCPOINT command 57
 - ROLLBACK statement
 - description 1039
 - error in IMS 772
 - in a stored procedure 571
 - TO SAVEPOINT clause 65
 - when to issue 56
 - with RRSAF 108
 - routines
 - inheriting special registers 548
 - row
 - selecting with WHERE clause 678
 - updating 670
 - updating current 729
 - updating large volumes 672
 - ROW CHANGE TIMESTAMP 694
 - ROW_NUMBER 685
 - row-level security 473
 - ROWID
 - data type 460
 - inserting in table 483
 - ROWID column
 - defining 658, 756
 - defining LOBs 463
 - inserting values into 658
 - using for direct row access 754
 - ROWID host variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
 - ROWID variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - Fortran 397
 - PL/I 409
 - rowset
 - deleting current 733
 - updating current 733
 - rowset cursor
 - closing 736
 - DB2 for z/OS down-level requester 878
 - declaring 732
 - end-of-data condition 732
 - example 751

- rowset cursor (*continued*)
 - multiple-row FETCH 733
 - opening 732
 - using 731
- RRSAF functions
 - summary of behavior 116
- RUN subcommand of DSN
 - return code processing 1017
 - running a program in TSO foreground 1017
- run time libraries, DB2I
 - background processing 987
 - EDITJCL processing 987
- running application program
 - CICS 1030
 - errors 1059
 - IMS 1030

S

- sample application
 - DRDA access 375
 - DRDA access with CONNECT statements 375
 - DRDA with three-part names 381
 - dynamic SQL 311
 - environments 1096
 - languages 1096
 - LOB 1094
 - organization 1094
 - phone 1094
 - programs 1092
 - project 1094
 - static SQL 311
 - stored procedure 1094
 - use 1092
 - user-defined function 1094
- sample applications 1071
 - databases 1089
 - storage 1088
 - storage groups 1089
 - structure 1088
- Sample applications
 - TSO 1097
- sample data 1071
- Sample data
 - joins 706
- sample program
 - DSN8BC3 357
 - DSN8BD3 307
 - DSN8BE3 307
 - DSN8BF3 403
 - DSN8BP3 427
- sample tables 1071
 - DSN8A10.ACT (activity) 1071
 - DSN8A10.DEMO_UNICODE (Unicode sample) 1082
 - DSN8A10.DEPT (department) 1072
 - DSN8A10.EMP (employee) 1074
 - DSN8A10.EMP_PHOTO_RESUME (employee photo and resume) 1077
 - DSN8A10.EMPPROJECT (employee-to-project activity) 1081
 - DSN8A10.PROJ (project) 1079
 - PROJECT (project activity) 1080
 - relationships 1083
 - storage 1088
 - views 1084
- samples
 - provided by DB2 1071
- SAVEPOINT statement 65
- savepoints 65
- scalar pointer host variable
 - declaring 300
 - referencing in SQL statements 298
- scrollable cursor
 - comparison of types 738
 - DB2 for z/OS down-level requester 878
 - dynamic
 - dynamic model 723
 - fetching current row 741
 - fetch orientation 737
 - retrieving rows 737
 - sensitive dynamic 723
 - sensitive static 723
 - sensitivity 738
 - static
 - creating delete hole 741
 - creating update hole 741
 - holes in result table 741
 - number of rows 739
 - removing holes 740
 - static model 723
 - updatable 723
- scrolling
 - backward through data 746
 - backward using identity columns 746
 - backward using ROWIDs 746
 - in any direction 738
 - ISPF (Interactive System Productivity Facility) 1049
- search condition
 - comparison operators 678
 - NOT keyword 678
 - SELECT statement 710
 - WHERE clause 678
- SELECT FROM DELETE statement
 - description 674
 - retrieving
 - multiple rows 674
 - with INCLUDE clause 674
- SELECT FROM INSERT statement
 - BEFORE trigger values 663
 - default values 663
 - description 663
 - inserting into view 663
 - multiple rows
 - cursor sensitivity 663
 - effect of changes 663
 - effect of SAVEPOINT and ROLLBACK 663
 - effect of WITH HOLD 663
 - processing errors 663
 - result table of cursor 663
 - using cursor 663
 - using FETCH FIRST 663
 - using INPUT SEQUENCE 663
 - result table 663
 - retrieving
 - BEFORE trigger values 663
 - default values 663
 - generated values 663
 - multiple rows 663
 - special registers 663
 - using SELECT INTO 663
- SELECT FROM MERGE statement
 - description 662
 - with INCLUDE clause 662

- SELECT FROM UPDATE statement
 - description 672
 - retrieving
 - multiple rows 672
 - with INCLUDE clause 663, 672
- SELECT INTO
 - using with host variables 184
- SELECT statement
 - AS clause
 - with ORDER BY clause 684
 - changing result format 1049
 - clauses
 - DISTINCT 682
 - EXCEPT 688
 - FROM 678
 - GROUP BY 692
 - HAVING 693
 - INTERSECT 688
 - ORDER BY 684
 - UNION 688
 - WHERE 678
 - derived column with AS clause 681
 - filtering by time changed 694
 - fixed-list 200
 - named columns 678
 - ORDER BY clause
 - derived columns 684
 - with AS clause 684
 - parameter markers 202
 - search condition 710
 - selecting a set of rows 722
 - subqueries 710
 - unnamed columns 681
 - using with
 - * (to select all columns) 678
 - column-name list 678
 - DECLARE CURSOR statement 727, 732
 - varying-list 202
- selecting
 - all columns 678
 - named columns 678
 - rows 678
 - some columns 678
 - unnamed columns 681
- semicolon
 - default SPUFI statement terminator 1041
 - embedded 1106
- sequence numbers
 - COBOL application program 357
 - Fortran 403
 - PL/I 427
- sequence object
 - creating 511
 - referencing 769
 - using across multiple tables 511
- server 68
- services
 - Message Queue Interface (MQI) 880
 - WebSphere MQ 879
- SET clause of UPDATE statement 670
- SET CURRENT PACKAGESET statement 951
- SET ENCRYPTION PASSWORD statement 477
- SET_CLIENT_ID (connection function of RRSAF)
 - language examples 137
 - syntax 137
- SET_ID (connection function of RRSAF)
 - language examples 136
- SET_ID (connection function of RRSAF) (continued)
 - syntax 136
- SET_REPLICATION (connection function of RRSAF)
 - language examples 140
 - syntax 140
- setting SQL terminator
 - DSNTIAD 1106
 - SPUFI 1047
- shortcut keys
 - keyboard xvi
- shredding XML documents from MQ messages 884
- SIGNAL statement
 - raising a condition 588
 - setting condition message text 589
- SIGNON (connection function of RRSAF)
 - language examples 124
 - program example 151
 - syntax 124
- SOME quantified predicate 712
- sort key
 - ORDER BY clause 684
 - ordering 684
- SOURCE precompiler option 933
- special register
 - behavior in stored procedures 572
 - behavior in user-defined functions and stored procedures 548
 - CURRENT PACKAGE PATH 950
 - CURRENT PACKAGESET 950
 - CURRENT RULES 972
- SPUFI
 - browsing output 1048
 - changed column widths 1049
 - CONNECT LOCATION field 1039
 - created column heading 1049
 - DB2 governor 1035
 - default values 1041
 - entering comments 1039
 - panels
 - allocates RESULT data set 1039
 - filling in 1035
 - format and display output 1048
 - previous values displayed on panel 1035
 - selecting on DB2I menu 1035
 - processing SQL statements 1035
 - setting SQL terminator 1047
 - specifying SQL statement terminator 1041
 - SQLCODE returned 1048
- SPUFI DEFAULTS panel 1043
- SQL (Structured Query Language)
 - checking execution 227
 - coding
 - dynamic 198
 - Fortran program 183
 - object extensions 512
 - cursors 722
 - dynamic
 - coding 193
 - sample C program 311
 - return codes
 - checking 228
 - handling 229
 - statement terminator 1106
 - string delimiter 988
 - syntax checking 876
 - varying-list 202

- SQL communication area (SQLCA)
 - description 228
 - using DSNTIAR to format 229
- SQL data types
 - compatibility with host language data types 180
- SQL linkage convention 619, 629
- SQL precompiler option 933
- SQL procedure
 - allowable statements 566
 - body 566
 - changing 598
 - conditions, handling 578
 - ignoring conditions 587
 - parameters 566
 - preparation using DSNTPSMP procedure 602
 - SQL variable 566
- SQL procedure processor (DSNTPSMP)
 - result set 611
- SQL procedure statement
 - CONTINUE handler 579
 - EXIT handler 579
 - handler 578
 - handling errors 578
- SQL procedures 565
 - creating versions 593
 - declaring cursors 577
 - nested compound statements 575
- SQL processing options/planning for 51
- SQL statement nesting
 - restrictions 720
 - stored procedures 720
 - user-defined functions 720
- SQL statement terminator
 - modifying in DSNTPE2 and DSNTPE4 1101, 1108
 - modifying in DSNTIAD 1106
 - modifying in SPUFI 1041
 - specifying in SPUFI 1041
- SQL statements
 - ALTER FUNCTION 517
 - checking for successful execution 173
 - CLOSE 200, 731, 736
 - COBOL program sections 357
 - coding REXX 183
 - comments
 - assembler 267
 - C 307
 - COBOL 357
 - Fortran 403
 - PL/I 427
 - REXX 439
 - CONNECT, with DRDA access 874
 - continuation
 - assembler 267
 - C 307
 - COBOL 357
 - Fortran 403
 - PL/I 427
 - REXX 439
 - CREATE FUNCTION 517
 - DECLARE CURSOR
 - description 727, 732
 - example 200, 202
 - DELETE
 - description 729
 - example 673
 - DESCRIBE 202
 - embedded 923
- SQL statements (*continued*)
 - error return codes 229
 - EXECUTE 221
 - EXECUTE IMMEDIATE 219
 - FETCH
 - description 729, 733
 - example 200
 - Fortran program sections 403
 - in application programs 159
 - INSERT 655
 - labels
 - assembler 267
 - C 307
 - COBOL 357
 - Fortran 403
 - PL/I 427
 - REXX 439
 - margins
 - assembler 267
 - C 307
 - COBOL 357
 - Fortran 403
 - PL/I 427
 - REXX 439
 - MERGE
 - example 661
 - OPEN
 - description 729, 732
 - example 200
 - PL/I program sections 427
 - PREPARE 221
 - RELEASE, with DRDA access 876
 - REXX program sections 439
 - SELECT
 - description 678
 - joining a table to itself 699
 - joining tables 695
 - SELECT FROM DELETE 674
 - SELECT FROM INSERT 663
 - SELECT FROM MERGE 662
 - SELECT FROM UPDATE 672
 - set symbols 267
 - UPDATE
 - description 729, 733
 - example 670
 - WHENEVER 233
- SQL table functions 522
- SQL terminator, specifying in DSNTPE2 and DSNTPE4 1101, 1108
- SQL terminator, specifying in DSNTIAD 1106
- SQL variable 566
- SQL-INIT-FLAG, resetting 357
- SQLCA (SQL communication area)
 - checking SQLCODE 232
 - checking SQLERRD(3) 228
 - checking SQLSTATE 232
 - checking SQLWARN0 228
 - description 228
 - DSNTIAC subroutine
 - assembler 267
 - C 307
 - COBOL 357
 - PL/I 427
 - DSNTIAR subroutine
 - assembler 239
 - C 307
 - COBOL 357

- SQLCA (SQL communication area) *(continued)*
 - DSNTIAR subroutine *(continued)*
 - Fortran 403
 - PL/I 427
 - sample C program 311
- SQLCA (SQL communications area)
 - assembler 253
 - C/C++ 273
 - COBOL 323
 - deciding whether to include 173
 - Fortran 395
 - PL/I 407
 - REXX 437
- SQLCODE
 - 923 975
 - 925 772
 - 926 772
 - +100 233
 - +802 240
 - values 232
- SQLCODE host variable
 - deciding whether to declare 173
- SQLDA
 - setting an XML host variable 202
 - XML column 202
- SQLDA (SQL descriptor area)
 - allocating storage 202, 733
 - assembler 173, 254
 - assembler program 202
 - C 202
 - C/C++ 173, 274
 - COBOL 173, 324
 - declaring 733
 - dynamic SELECT example 202
 - for LOBs and distinct types 202
 - Fortran 173, 396
 - multiple-row FETCH statement 733
 - no occurrences of SQLVAR 202
 - OPEN statement 200
 - parameter markers 202
 - PL/I 173, 202, 408
 - requires storage addresses 202
 - REXX 173, 437
 - setting output fields 733
 - storing parameter marker information 226
 - varying-list SELECT statement 202
- SQLERROR clause of WHENEVER statement 233
- SQLN field of SQLDA 202
- SQLRULES, option of BIND PLAN subcommand 972
- SQLSTATE
 - "01519" 240
 - "2D521" 772
 - "57015" 975
 - values 232
- SQLSTATE host variable
 - deciding whether to declare 173
- SQLSTATES
 - web service consumer 912
- SQLVAR field of SQLDA 202
- SQLWARNING clause of WHENEVER statement 233
- SSID (subsystem identifier), specifying 987
- static SQL
 - C/C++ application program
 - examples 311
 - description 193
 - host variables 194
 - sample C program 311
- statistics
 - real-time
 - stored procedure 842
- STDSQL precompiler option 933
- storage
 - acquiring
 - retrieved row 202
 - SQLDA 202
 - addresses in SQLDA 202
- storage groups
 - for sample applications 1089
- storage shortages
 - when calling stored procedures 803
- stored procedure
 - abend 791
 - accessing CICS 639
 - accessing IMS 639
 - accessing non-DB2 resources 639
 - accessing transition tables 551
 - authorization to run 791
 - CALL statement 791
 - calling from a REXX procedure 797
 - calling from an application 791
 - COMMIT statement 571
 - compatible data types 797
 - creating 558
 - creating external stored procedure 615
 - cursors 571
 - Data types 646
 - defining parameter lists 619
 - DSNACCOR 842
 - example 562
 - invoking from a trigger 501
 - languages supported 569
 - linkage conventions 619
 - preparation 558
 - real-time statistics 842
 - reentrant 643
 - returning non-relational data 641
 - returning result set 641
 - ROLLBACK statement 571
 - running multiple instances 803
 - types 558
 - use of special registers 572
 - using host variables with 562
 - using temporary tables in 641
 - WLM_REFRESH 816
 - writing 569
 - writing in REXX 650
- stored procedure result sets
 - receiving in a program 807
- stored procedures
 - >DSNACCOX 811
 - ADMIN_COMMAND_DB2 811
 - ADMIN_COMMAND_DSN 811
 - ADMIN_COMMAND_UNIX 811
 - ADMIN_DS_BROWSE 811
 - ADMIN_DS_DELETE 811
 - ADMIN_DS_LIST 811
 - ADMIN_DS_RENAME 811
 - ADMIN_DS_SEARCH 811
 - ADMIN_DS_WRITE 811
 - ADMIN_INFO_HOST 811
 - ADMIN_INFO_SSID 811
 - ADMIN_JOB_CANCEL 811
 - ADMIN_JOB_FETCH 811
 - ADMIN_JOB_QUERY 811

- stored procedures (*continued*)
 - ADMIN_JOB_SUBMIT 811
 - ADMIN_TASK_ADD 811
 - ADMIN_TASK_REMOVE 811
 - ADMIN_UTL_SCHEDULE 811
 - ADMIN_UTL_SORT 811
 - calling other programs 643
 - creating native SQL procedures 572
 - DB2-supplied 811
 - debugging 1052
 - debugging with the Unified Debugger 1056
 - description 559
 - DSNACCOR 811
 - DSNACICS 811, 825
 - DSNAEXP 811
 - DSNAHVPM 811
 - DSNAIMS 811, 833
 - DSNAIMS2 811, 838
 - DSNLEUSR 811
 - DSNTBIND 811
 - DSNTPSMP 811
 - DSNUTILS 811
 - DSNUTILU 811
 - DSNWSPM 811
 - DSNWZP 811
 - from command line processor 1028
 - GET_CONFIG 811
 - GET_MESSAGE 811
 - GET_SYSTEM_INFO 811
 - inheriting special registers 548
 - migrating external SQL to native SQL 596
 - package collection ID 643
 - packages for nested routines 643
 - parameter list 561
 - passing large output parameters 796
 - recording debugging messages 1058
 - running concurrently 805
 - SQLJ.ALTER_JAVA_PATH 811
 - SQLJ.DB2_INSTALL_JAR 811
 - SQLJ.DB2_REMOVE_JAR 811
 - SQLJ.DB2_REPLACE_JAR 811
 - SQLJ.DB2_UPDATEJARINFO 811
 - SQLJ.INSTALL_JAR 811
 - SQLJ.REMOVE_JAR 811
 - SQLJ.REPLACE_JAR 811
 - syntax for invoking from command line processor 1028
 - WLM_REFRESH 811
 - XDBDECOMPXML 811
 - XSR_ADDSCHEMADOC 811
 - XSR_COMPLETE 811
 - XSR_REGISTER 811
 - XSR_REMOVE 811
- storm drain effect 156
- string
 - data type 460
- structure array host variable
 - declaring 300
 - referencing in SQL statements 298
- subquery
 - basic predicate 712
 - conceptual overview 710
 - correlated
 - DELETE statement 716
 - description 714
 - example 714
 - UPDATE statement 716
 - DELETE statement 716

- subquery (*continued*)
 - description 710
 - EXISTS predicate 712
 - IN predicate 712
 - quantified predicate 712
 - referential constraints 718
 - restrictions with DELETE 718
 - UPDATE statement 716
- subsystem
 - identifier (SSID), specifying 987
- subsystem parameters 805
- summarizing group values 692
- SWITCH TO (connection function of RRSAF)
 - language examples 121
 - syntax 121
- SYNC call, IMS 61
- synchronization call abends 772
- SYNCPPOINT command of CICS 57
- syntax diagram
 - how to read xvii
- SYSIBM.MQPOLICY_TABLE
 - column descriptions 884
- SYSIBM.MQSERVICE_TABLE
 - column descriptions 884
- SYSLIB data sets 978
- SYSPRINT precompiler output
 - options section 1061
 - source statements section, example 1061
 - summary section, example 1061
 - symbol cross-reference section 1061
 - used to analyze errors 1061
- SYSTEM output to analyze errors 1061

T

- table
 - altering
 - changing definitions 478
 - using CREATE and ALTER 240
 - copying from remote locations 871
 - declaring in a program 160
 - deleting rows 673
 - dependent, cycle restrictions 472
 - displaying, list of 677
 - DROP statement 484
 - filling with test data 1034
 - incomplete definition of 484
 - inserting multiple rows 657
 - inserting single row 655
 - loading, in referential structure 471
 - merging rows 661
 - populating 1034
 - referential structure 471
 - retrieving 722
 - selecting values as you delete rows 674
 - selecting values as you insert rows 663
 - selecting values as you merge rows 662
 - selecting values as you update rows 672
 - temporary 481
 - updating rows 670
 - using three-part table names 871
- table and view declarations
 - including in an application program 169
- table and view declarationsgenerating with DCLGEN 161
- table declarations
 - adding to libraries 170

- table locator
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - PL/I 409
- table space
 - not logged
 - recovering 66
- table spaces
 - for sample applications 1090
- tables
 - creating for data integrity 468
 - supplied by DB2
 - DSN_FUNCTION_TABLE 784
- TCB (task control block)
 - capabilities with CAF 77
 - capabilities with RRSAF 108
- temporary table
 - advantages of 481
 - working with 481
- terminal monitor program (TMP) 1017
- TERMINATE IDENTIFY (connection function of RRSAF)
 - language examples 144
 - program example 151
 - syntax 144
- TERMINATE THREAD (connection function of RRSAF)
 - language examples 143
 - program example 151
 - syntax 143
- TEST command of TSO 1064
- test environment, designing 1017
- test tables 1031
- test views of existing tables 1031
- TIME precompiler option 933
- time that row was changed
 - determining 770
- TMP (terminal monitor program)
 - DSN command processor 1017
 - running under TSO 1029
- transition table, trigger 493
- transition variable, trigger 493
- TRANSLATE (connection function of CAF)
 - description 86
 - language example 96
 - program example 100
 - syntax 96
- TRANSLATE (connection function of RRSAF)
 - language examples 146
 - syntax 146
- translating requests into SQL 240
- trigger
 - activation order 506
 - activation time 493
 - cascading 505
 - coding 493
 - data integrity 508
 - delete 493
 - description 493
 - FOR EACH ROW 493
 - FOR EACH STATEMENT 493
 - granularity 493
 - insert 493
 - interaction with constraints 506
 - interaction with security label columns 508
 - invoking stored procedure 501
 - invoking user-defined function 501
 - naming 493
 - trigger (*continued*)
 - parts example 493
 - parts of 493
 - passing transition tables 501
 - subject table 493
 - transition table 493
 - transition variable 493
 - triggering event 493
 - update 493
 - using identity columns 493
 - with row-level security 508
- troubleshooting
 - errors for output host variables 179
- TRUNCATE 673
 - example 673
- truncated
 - determining value of output host variable 186
- TSO
 - CLISTs
 - calling application programs 1030
 - running in foreground 1030
 - TEST command 1064
 - TWOPASS precompiler option 933

U

- Unicode
 - data, retrieving from DB2 for z/OS 202
 - sample table 1082
- Unified Debugger
 - debugging stored procedures 1056
 - setting up 1056
- UNION
 - eliminating duplicate rows 689
 - keeping duplicate rows with ALL 691
- UNION clause
 - columns of result table 688
 - combining SELECT statements 688
- UNIQUE clause 474
- unit of work
 - CICS 57
 - completion
 - open cursors 725
 - description 55
 - IMS 61
 - TSO 56
 - undoing changes within 65
- Universal language interface 153
- updatable cursor 727
- UPDATE statement
 - correlated subqueries 716
 - description 670
 - positioned
 - FOR ROW n OF ROWSET 733
 - restrictions 729
 - WHERE CURRENT clause 729, 733
 - SET clause 670
- updating
 - during retrieval 708
 - large volumes 672
- updating data
 - by using host variables 189
- USER special register
 - value in INSERT statement 460
 - value in UPDATE statement 670
- user-defined function
 - Debug Tool 1050

- user-defined function (UDF)
 - abnormal termination 554
 - accessing transition tables 551
 - ALTER FUNCTION statement 517
 - authorization ID 779
 - call type 535
 - casting arguments 788
 - characteristics 523
 - coding guidelines 525
 - CREATE FUNCTION statement 517
 - data type promotion 780
 - DBINFO structure 537
 - definer 520
 - defining 523
 - description 520
 - diagnostic message 534
 - DSN_FUNCTION_TABLE 783
 - example
 - external scalar 517, 556
 - external table 517
 - function resolution 780
 - overloading operator 517
 - sourced 517
 - SQL 517
 - function resolution 780
 - host data types
 - assembler 528
 - C 528
 - COBOL 528
 - PL/I 528
 - implementer 520
 - implementing 521
 - indicators
 - input 534
 - result 534
 - inheriting special registers 548
 - invoker 520
 - invoking 777
 - invoking from a trigger 501
 - invoking from predicate 777
 - main program 525
 - multiple programs 547
 - naming 534
 - nesting SQL statements 720
 - parallelism considerations 525
 - parameter conventions 528
 - assembler 540
 - C 541
 - COBOL 544
 - PL/I 546
 - preparing 554
 - reentrant 547
 - restrictions 525
 - samples 557
 - scratchpad 535, 555
 - scrollable cursor 777
 - setting result values 533
 - simplifying function resolution 779
 - specific name 534
 - steps in creating and using 520
 - subprogram 525
 - table locators
 - assembler 551
 - C 552
 - COBOL 553
 - PL/I 554
 - testing 1050

- user-defined function (UDF) (*continued*)
 - types 520
- user-defined functions
 - SOAPHTTPNC 911
 - SOAPHTTPNV 911
- USING DESCRIPTOR clause
 - EXECUTE statement 202
 - FETCH statement 202
 - OPEN statement 202

V

- VALUES clause, INSERT statement 655
- varbinary host variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - PL/I 409
- varbinary host variable array
 - C/C++ 287
 - PL/I 415
- variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - declaring in SQL procedure 566
 - Fortran 397
 - PL/I 409
- variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
- version
 - changing for SQL procedure 598
- version of a package 945
- VERSION precompiler option 933, 945
- versions
 - procedures 593
- view
 - contents 486
 - declaring in a program 160
 - description 485
 - dropping 487
 - identity columns 485
 - join of two or more tables 486
 - referencing special registers 485
 - retrieving 722
 - summary data 486
 - union of two or more tables 486
 - using
 - deleting rows 673
 - inserting rows 655
 - updating rows 670

W

- web service consumer
 - SQLSTATEs 912
- WebSphere MQ
 - APIs 879
 - description 879
 - interaction with DB2 879
 - message handling 879
 - Message Queue Interface (MQI) 880
 - messages 879

- WHENEVER statement
 - assembler 267
 - C 307
 - COBOL 357
 - CONTINUE clause 233
 - Fortran 403
 - GO TO clause 233
 - NOT FOUND clause 233, 729
 - PL/I 427
 - specifying 233
 - SQL error codes 233
 - SQLERROR clause 233
 - SQLWARNING clause 233
- WHERE clause
 - SELECT statement
 - description 678
 - joining a table to itself 699
 - joining tables 695
- WITH clause
 - common table expressions 488
- WITH HOLD clause
 - and CICS 725
 - and IMS 725
 - DECLARE CURSOR statement 725
 - restrictions 725
- WITH HOLD cursor
 - effect on dynamic SQL 221
- WLM_REFRESH stored procedure
 - description 816
 - option descriptions 817
 - sample JCL 818
 - syntax diagram 817
- WLM_SET_CLIENT_INFO procedure 820
- write-down privilege 508

- XMLQUERY 681
 - description 681
 - example 681
- XPath 681
 - XPath contexts 681
- XPath contexts
 - XMLEXISTS 771
- XPath expressions 681
- XREF precompiler option 933
- XSR_COMPLETE stored procedure 866
- XSR_REGISTER
 - register XML schema 863

X

- XML data
 - embedded SQL applications 241
 - retrieving from tables, embedded SQL applications 248
 - selecting 681
 - updating, embedded SQL applications 246
- XML file reference variable
 - assembler 255
 - C/C++ 275, 287
 - COBOL 326, 335
 - PL/I 409, 415
- XML host variable
 - SQLDA 202
- XML host variable array
 - C/C++ 287
 - COBOL 335
 - PL/I 415
- XML schema registration
 - XSR_ADDSCHEMADOC stored procedure 865
 - XSR_COMPLETE stored procedure 866
 - XSR_REMOVE stored procedure 868
- XML values
 - fetching 742
- XML variable
 - assembler 255
 - C/C++ 275
 - COBOL 326
 - PL/I 409
- XMLEXISTS 771
 - description 771
 - example 771



Product Number: 5605-DB2
5697-P31

Printed in USA

SC19-2969-10



Spine information:

DB2 10 for z/OS

Application Programming and SQL Guide

